NAT'L INST. OF STAND & TECH
A11106 039324

NOV 15 1985

NBSIR 85-3250

Characteristics and Functions of Software Engineering Environments

REFERENCE NBS PUBLICATIONS

Raymond C. Houghton, Jr.

Computer Science Department
Duke University

Dolores R. Wallace

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Institute for Computer Sciences and Technology

September 1985

U.S. DEPARTMENT OF COMMERCE
L BUREAU OF STANDARDS

QC
100
.U56
85-3250
1985
C.3

NBSIR 85-3250

CHARACTERISTICS AND FUNCTIONS OF SOFTWARE ENGINEERING ENVIRONMENTS

Raymond C. Houghton, Jr.

Computer Science Department
Duke University

Dolores R. Wallace

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Institute for Computer Sciences and Technology

September 1985

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

Characteristics and Functions of Software Engineering Environments

Raymond C. Houghton, Jr. and Dolores R. Wallace

ABSTRACT

As part of the program to provide information to Federal agencies on software tools for improving quality and productivity in software development and maintenance, data was collected on software engineering environments. Software engineering environments surround their users with software tools necessary for systematic development and maintenance of software. The purpose of this report is to characterize software engineering environments by type and by their relationship to the software life cycle and by their capabilities, limitations, primary users, and levels of support. This report provides examples of existing software engineering environments that are available commercially or in research laboratories.

KEYWORDS

framing environments; human factors; life cycle coverage; programming environments; software analysis; software engineering; software engineering environments; software support; software tools.

FOREWORD

Under the Brooks Act, the National Bureau of Standards Institute for Computer Sciences and Technology (ICST) promotes the cost effective selection, acquisition, and utilization of automatic data processing resources within Federal agencies. ICST efforts include research in computer science and technology, direct technical assistance, and the development of Federal standards for data processing equipment, practices, and software.

ICST has published several documents on software tools as part of this responsibility and the growing recognition that the use of software tools and software engineering environments can reduce the effort necessary to develop and maintain computer software. The guidance is designed to assist Federal agencies in automating and standardizing their software development and maintenance projects.

This report presents the results of the analysis of data and experience accumulated on software engineering environments. It characterizes environments and describes their features to enable readers to gain an understanding of how environments can aid software development and maintenance process. Future ICST documents will provide guidance in selecting and using software engineering environments.

Certain commercial products are identified in this paper for clarification of specific concepts. In no case does such identification imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the material identified is necessarily the best for the purpose.

TABLE OF CONTENTS

1.0	INTRODUCTION	1
2.0	ENVIRONMENTS AND LIFE CYCLE RELATIONSHIPS	2
2.1	Life Cycle Defined	2
2.2	Life Cycle Coverage	3
2.3	Programming Environments	4
2.4	Framing Environments	5
2.5	General Environments	6
3.0	INTEGRATION	7
3.1	Levels of Integration	7
3.2	User Interface	9
3.3	Database Interface	10
3.4	Binding the Tools in an Environment	10
4.0	HUMAN FACTORS	11
4.1	On-Line Assistance	11
4.1.1	Command Assistance	11
4.1.2	HELP Assistance	12
4.1.3	Error Assistance	12
4.1.4	On-line Tutor	12
4.1.5	On-line Documentation	12
4.1.6	Other Types of User Assistance	12
4.2	Quality Factors	13
4.2.1	Robustness	13
4.2.2	Pitfalls	14
4.2.3	Other Quality Factors	14
4.3	Implementation Techniques	14
4.3.1	Query-in-Depth	14
4.3.2	Contextual Assistance	15
4.3.3	Natural Language	15
4.3.4	Simulation and Prototyping	15
4.3.5	Contextual Mode Switching	16
5.0	ANALYSIS AND SOFTWARE QUALITY	16
5.1	Static Analysis	16
5.2	Dynamic Analysis	17
5.3	Management	18
5.4	Underlying Analysis Features	19
6.0	SUPPORT FOR DIFFERENT TYPES OF USERS	19
6.1	The Manager	20
6.2	The Designer	21
6.3	The Programmer	21
6.4	The Analyst	21
6.5	The Documentation Editor	22
6.6	The Librarian	23
6.7	The Maintainer	23
7.0	SUPPORT FOR APPLICATION	23
7.1	Systems Development	23
7.2	Embedded Systems	24

7.3	Information Systems	24
7.4	Data Processing Applications	24
7.5	Security-Critical Applications	25
8.0	HARDWARE SUPPORT	25
8.1	Hardware Selection Issues	26
8.2	Benefits Gained by Selecting and Dedicating Hardware	26
8.3	Workstations	26
9.0	LEVELS OF SUPPORT	27
9.1	Levels of Support Based on Project Size	27
9.2	Levels of Support Based on Capability	27
9.3	Levels of Support Based on User Priorities	31
9.4	Generic Support	31
10.0	CONCLUSION	31
11.0	REFERENCES	33

1.0 INTRODUCTION

Although the IEEE Standard Glossary of Software Engineering Terminology [IEEE729] does not define a software engineering environment, it does provide a definition for a programming support environment:

> An integrated collection of tools accessed via a single command language to provide programming support capabilities throughout the software life cycle. The environment typically includes tools for design, editing, compiling, loading, testing, configuration management, and project management.

According to Funk & Wagnall's Standard Desk Dictionary, environment is "surroundings". Thus, a programming environment (or programming support environment) surrounds the user with the tools needed to program. Similarly, a software engineering environment surrounds the user with the tools needed to systematically develop and maintain software. Software includes programs, data (such as test input and output data), and documentation. Therefore, a programming environment is a software engineering environment with limited scope. A typical operating system might be referred to as a software engineering environment, but its scope may be even more limited.

Software engineering environments are popular research topics because the technology used to develop them can draw upon many aspects of software engineering and computer science, including systems design, artificial intelligence, language design and processing, information handling, networking and communications, and correctness issues. As a result, software engineering environments are a consistent topic at conferences and workshops. Some recent examples [1] are:

- The NBS Programming Environment Workshop [NBS78]
- 2 sessions at the 5th International Conference on Software Engineering [ICSE81] and several demonstrations [NBS80]
- 2 sessions at the 6th International Conference on Software Engineering [ICSE82]
- 4 sessions at SoftFair (A Conference on Software Development Tools, Techniques, and Alternatives) [Soft83]
- 2 sessions at the 7th International Conference on Software Engineering [ICSE84]
- The ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments [ACM84]

There is a proven cost savings when software engineering environments are used. In an analysis of 63 software development projects (ranging from business applications to process control applications), Barry Boehm [Boeh81] found a good correlation between the low, nominal, and high use of tools and the effort necessary to develop software. Based on this analysis, use of tools that would normally be found in a software engineering environment can reduce the development effort 9-17%.

The use of a software engineering environment interacts with other factors that affect software development. Other analyses by Boehm showed that (1) the encouraged use of modern programming practices that would result from using an environment can reduce development effort 9-18% and (2) the improvement in turnaround time from using an environment can reduce development effort 13%. All factors combined, the use of software engineering environments can reduce development effort 28-41% [Boeh81].

A General Accounting Office (GAO) report [GAO80] endorses the use of software development automation and concludes that software tools (such as those found in software engineering environments) can offer the Federal Government:

(1) Better management control of computer software development, operation, maintenance, and conversion,

(2) Lower costs for computer software development, operation, maintenance, and conversion,

(3) Feasible means of inspecting both contractor-developed and in-house-developed computer software for such quality indications as conformance to standards and thoroughness of testing.

The GAO report was based primarily on site visits to various Government agencies.

The purpose of this report is to characterize software engineering environments by type, by their relationship to the software lifecycle, and by their capabilities, limitations, primary users, and levels of support. This report provides examples of existing software engineering environments. The sections that follow address the types of environments and their relationships to the software life cycle (Section 2); the features that affect their development (Sections 3 and 4); the features they provide (Section 5); the orientations they can have (Sections 6 and 7); the hardware they can use (Section 8); and the levels of support they can provide (Section 9).

[1] NBS/ICST was a cosponsor of all of these workshops and conferences.

2.0 ENVIRONMENTS AND LIFE CYCLE RELATIONSHIPS

One of the most recognized aspects of software engineering is what is commonly referred to as the software life cycle. The software life cycle is a model of the software development and maintenance process. This process includes a series of transformations and activities from initiation to retirement, resulting in products as outcomes of the transformations and activities, where transformations may be iterated at any time in the life cycle. Software maintenance, for example, may include iteration throughout the cycle. Software products that have reached the retirement stage are often redeveloped unless there is no longer a need for the application (for which the products were originally engineered.) The redevelopment may include all of the activities and transformations, and iterations to achieve the desired outcomes. Because of this phenomenon, the term "cycle" is used to refer to the "life" of software products.

Because software products undergo these transitions, software engineering environments attempt to provide support that will make the transitions smoother, less costly, and less error-prone.

2.1 Life Cycle Defined

Although the term "phase" is frequently used to frame the activities and transformations of the software life cycle process and the documentation of those activities and transformations, there is no standard set of phases for the software life cycle. This document uses a general framework in which to group the software development and maintenance process according to similar transformations and activities, and to define the software products as outcomes of these transformations and activities. This document uses the following description of the software development and maintenance process as a frame of reference for the characterizations and features of software engineering environments:

a. Initiation

- The need for a software system is conceived.
- A general definition of the requirements is established.
- Feasibility studies are performed.
- A development plan is defined.
- Cost/benefit analysis is performed.

b. Definition

- A description of the system is developed including functional and data requirements.
- The resource constraints (cost, performance, hardware, etc.) are established.
- The Validation, Verification, and Testing (VV&T) plan is developed.
- Requirements-based test cases are generated.

c. High Level Design

- The system architecture (major parts of the system and data flow between the parts) is defined.
- Basic algorithms and major data representations are established.
- Design-based test scenarios are generated.
- Verification that design satisfies the requirements is performed.

d. Detailed Design

- The major parts of the system are further defined.
- Precise algorithms and detailed data representations are established.
- Design-based functional test data are generated.
- Test support software is developed.
- Detailed design is verified.

e. Programming

- The detailed design is implemented into computer code.
- Pieces of code are tested and integrated with other pieces of code.
- System testing, performance evaluation, and acceptance testing are performed.

f. Operations and Maintenance

- The software system is used.
- Changes are made to the software as additional requirements are made or problems are found (cycle to a. Initiation)
- Regression Testing is performed (see Section 5.2)

g. Retirement

- The software system has minimal usefulness.
- Changes to the system begin to obscure the original requirements

and design.
- The system is expensive to maintain.
- The system is retired, or
- To repeat activities, transformations, the software process cycles back to (a. Initiation).

During software maintenance, changes are made to the software by re-performing the process of software development. Maintenance is a redevelopment process repeating the activities and transformations indicated in (a. Initiation - e. Programming) each time there are any new requirements for the software. For example, changes affecting the requirements cause a cycle back to initiation, where a feasibility study of the change and a development plan for the change should be performed.

Although the above life cycle has only one explicit cycle, it is important to note that there are many implicit cycles. The most notable stems from the discovery of a needed change during operation and maintenance. Verification activities that lead to error discovery in software design descriptions lead to iteration of previous activities of the process. Similarly, error discovery or new requirements at any point of software development or maintenance lead to implicit cycles requiring iteration throughout the life cycle.

Since changes and problems cause repetitions of previous phases, it is easy to understand why implicit cycles increase the cost of software. On the other hand, perfection is difficult to attain, so a certain amount of implicit cycling must be expected. As mentioned earlier, it is important for a software engineering environment to support the transitions from one phase to another. It is also important for an environment to support implicit cycling.

A software engineering environment supports the development of many software products. The term software refers not only to code but also to documentation. Software products therefore include the various forms of code and documentation. Although there is some disagreement as to which software products are formally or informally required in the software life cycle (and, in some cases, when they should be required), the following is a list compiled from Federal Information Processing Standards Publications [FIPS38, 64,101,106].

a. Initiation

- General definition of the requirements.
- Feasibility Study
- Development Plan
- Cost/Benefit Analysis

b. Definition

- Functional Requirements Document
- Data Requirements Document
- VV&T Plan

c. High Level Design

- System/Subsystem Specification
- VV&T Plan

d. Detailed Design

- Program Specification
- Data Base Specification
- Test Plan

e. Programming

- Users Manual
- Operations Manual
- Program Maintenance Manual
- Test Analysis Report

f. Operations and Maintenance

- Formal Change Request
- Updated Documentation

Since all these documents are not necessary for all development efforts, the FIPS PUBs provide flexibility in determining what documentation is required. For the same reason, software engineering environments must provide the same flexibility.

2.2 Life Cycle Coverage

One can see from the diverse list of products and activities that occur during the software life cycle, that it is difficult, if not impossible, for an environment to provide complete support for all software engineering activities from initiation to retirement.

An environment may support all phases of software engineering, in several ways, with varying degrees of support. An environment may completely support a methodology that is well defined for all phases and has features that support automation. Or, for some phases it may contain general tools (such as text processors, editors, and compilers) along with some methodology-dependent tools for some activities of software development and maintenance. Finally, an environment may support all the activities of all phases with only general tools. A recent survey of software engineering methodologies [Porc83] supports the claim that no methodology exists for the first case. Of 24 methodologies surveyed, only six

can be applied throughout the life cycle. Three of the six have automation support for some activities of the life cycle, but none of the methodologies has a supporting environment for all activities.

Therefore, it is fairly safe to conclude that environments that provide full support for the entire life cycle must contain general tools for some of the life cycle phases. Because the activities that occur in latter stages of the life cycle are so different from the earlier stages, environments tend to concentrate on either one or the other.

Environments that concentrate on the latter stages are typically called "programming environments". Those that concentrate on earlier stages, where the system is "framed" by its requirements and design, are referred to as "framing environments". It is important, however, to note that because of implicit looping in the life cycle, framing environments can provide some support for all "phases" of the life cycle; but they don't support all activities in each of the phases. For example, a framing environment supports the operation and maintenance phase when it allows changes in requirements resulting from errors detected during operation. However, it is not likely to support regression testing; that activity is more likely to be found in a programming environment.

A third type of environment may be described as a "general" environment. These environments contain basic tools that support all phases of the life cycle. They typically support more than one programming language. They usually have more advanced tools that support some of the early phases of the life cycle. Environments that fall under this class provide a "toolbox" of supporting capabilities that the user can apply at his or her discretion.

In the sections in this chapter, the typical characteristics of programming, framing, and general environments are discussed by presenting several examples of each. The example discussions are limited to the life cycle characteristics of the environments. Further discussion of other characteristics of many of these environments does occur in later chapters.

2.3 Programming Environments

Programming environments concentrate on the activities that are performed during the latter part of the life cycle. These environments provide features that are oriented toward a programming language (usually a high level language) with particular emphasis on coding, debugging, and testing of programs.

The programming environment that has received the most attention recently is the Ada [2] Programming Support Environment (APSE). Buxton lists the requirements for the APSE in the STONEMAN report [Buxt80]. Although the requirements emphasize full life cycle coverage, the APSE's that are currently or nearly available primarily support the programming phase. It is anticipated that future Ada environments will provide full life cycle coverage for a well defined DoD-wide software development methodology [Free83] which is currently being defined. The APSE is also discussed in Sections 3.1 and 9.3.

Examples of four other programming environments are Toolpack [Cowe83], Interlisp [Teit81], Cedar [Teit84], and Smalltalk-80 [Love83]. Toolpack is a portable, Fortran-oriented programming environment that is currently under development. Interlisp is an environment that is very much language dependent and is intended for use by Lisp experts. The Cedar environment emphasizes the use of parallel operation, multiple windows on a screen, and user interaction with a mouse pointing device. Cedar supports the use of an "industrial strength" Pascal-like programming language called the Cedar Programming Language. Smalltalk-80 is a single-user, single-language environment that supports object-oriented programming using the Smalltalk language.

A very promising area of research in programming environments is incremental development. Normally program construction is done sequentially using the editor, compiler, linker, and loader. Each sequence is completed before the next one is started and the objects passed from one sequence to the next are syntactically complete, executable programs. Incremental development synthesizes these sequences so that they do not have to be completed and the objects passed can be program fragments.

For example, syntax-directed editing combines features of a compiler with the editor. As a program is entered, lexical and syntactical analyzers process and direct the statements entered. The result is that much of the compiling is done as each statement is entered. Two examples of syntax-directed programming environments are the Cornell Program Synthesizer (CPS) [Teit81a] and POE [Fisc84]. CPS generates PL/CS programs; PL/CS is an instructional dialect of PL/I. POE generates Pascal programs.

Figure 2.1 shows the template for a conditional statement that is generated by CPS. "Condition" and "statement" are placeholders that a user is expected to fill in. CPS has templates for each statement type; thus, the user enters a

[2] Ada is a registered trademark of the U.S. Government Ada Joint Program Office

syntactically correct program that is partially compiled as it is entered.

IF (condition)
THEN statement
ELSE statement

Figure 2.1 CPS's Template for a Conditional Statement

A step further toward incremental development is obtained when the environment allows incremental compilation. That is, a user is allowed to execute program fragments. An example of an environment with this capability is Magpie [Deli84]. Magpie generates Pascal programs. It allows the user to execute a subset of the Pascal statements that have been entered. Magpie also provides syntax-directed editing.

Incremental development can also be extended further into design. An example of an environment that provides this capability is the CDL2 Laboratory. Each source unit (program fragment) is characterized by one of the following development stages:

modified - recently edited and changed.

compatible - the interface fits into the environment, but the unit may still contain internal ambiguities or conflicts; i.e., it is not yet consistent.

consistent - compatible with the outside, and free of internal ambiguities and conflicts.

complete - consistent and fully constructed; precondition for coding.

coded - complete and code generated.

Thus, a user can design program structures that are compatible and consistent, but not complete or coded.

2.4 Framing Environments

Framing environments concentrate on the early stages in the life cycle. Studies have shown that if software can be well defined (i.e., eliminate as many errors as possible) in the early stages of software development, a tremendous savings of resources can be realized in the later stages of the life cycle [Boeh81]. Therefore, it is not surprising that environments have been developed that concentrate on these more crucial stages of software development. Framing environments include the following examples.

SDS

(the Software Development System) [Alfo81] is a methodology and support environment for developing very large, complex, real-time systems. The methodology consists of four major tasks:

(1) Data Processing Systems Engineering (DPSE) - translate systems objectives into a consistent, complete set of subsystem functional and performance requirements (uses techniques based on verification graphs, petri nets, finite state machines, and graph models of decomposition for expressing requirements),

(2) Software Requirements Engineering Methodology (SREM) - express functional and performance requirements as a graph model in Requirements Statement Language (RSL) and analyze with the Requirements Engineering Validation System (REVS),

(3) Process Design Engineering - translate requirements into a process design language, verify design, and evolve the design into code,

(4) Verification and Validation - perform at all stages.

The SDS environment contains tools that start at the top system level and emphasize support for the decomposition and allocation of functional and performance requirements.

SARA

(System Architects' Apprentice) [Estr78] is a computer-aided design environment which supports a structured, multi-level design methodology for the design of hardware or software systems. It comprises a number of language processors and tools for assisting designers using the SARA methodology, together with a user-interface capability for assisting designers using the SARA system. The fundamental tool in the SARA environment is the graph model simulator [Razo80]. During top-down refinement of a design, the simulator is used to test consistency between the levels of abstraction.

DREAM

Design Realization, Evaluation and Modeling System [Ridd81] is a software engineering environment that is oriented to the development of concurrent systems using DREAM Design Notation (DDN). DDN is a language that can be used to model a total system including hardware, software, and other processes. The model reflects the externally observable characteristics of a

system and is an adequate basis for preparing implementation plans. The DREAM system tools include a data base core that stores DDN fragments, bookkeeping tools (entry and retrieval), and decision-making tools for paraphrasing (a re-structured presentation), extraction (simulation), and consistency checking.

There has been much discussion [ACM82] on the benefits of prototyping. Prototyping provides a working model of a system and provides immediate feedback from potential users. If the working model is developed early in the life cycle, it can short-circuit many of the life cycle's implicit loops. An example of an environment that supports the development of prototypes is the Unified Support Environment:

USE
: (Unified Support Environment) [Wass83] supports the User Software Engineering methodology. The methodology involves users early in development and addresses user interactions with information systems. The tools in the environment include: the Troll relational database (underlies and is used by other tools), RAPID (rapid prototyping tool oriented to the development of information systems), PLAIN (a procedural language oriented to the development of information systems), Focus (screen-oriented editor and browser), and IDE (a software management and control tool).

It is important to note that in all of the above examples of framing environments, a methodology for the use of the environments has been stressed. Because framing environments stress the development of specifications and models, the techniques used are very specific and well defined. Consequently, the environment contains tools that are very specialized.

2.5 General Environments

General environments are environments that do not necessarily fit in the programming or framing environment category. They contain basic tools (editors, text processors, etc.) that support all phases of the life cycle, but they may also have advanced tools only for certain phases. They usually don't require a specific software methodology, but can be adapted to most methodologies. Examples of general environments include the following:

ARGUS
: [Stuc83] is a general environment that contains specific tools for software design, management, and testing. Six toolboxes are available: the management toolbox (scheduling tools, action item tracking tool, electronic spread sheet, and phone list update and retrieval system); the designer's toolbox (software design capabilities with a graphics/forms based approach); the programmer's toolbox (language-based, project-specific code template capabilities provided by a customizable editor and language specific syntax generation macros); the verifier's toolbox (analysis tools); the configuration toolbox (general editing selection and formatting tools); and the utility toolbox (general editing and communication tools). Argus is also discussed in Sections 3.2 and 6.

SPS-1
: (Software Productivity System) [Boeh84] is a prototype environment that will eventually lead to the development of a full production version. The components of SPS-1 include: a master project database (composed of a hierarchical file system, a source code control system, and a relational database), general utilities (including a screen editor, forms package, and a report writer), office automation and project support (including a tool catalog, mail system, text editor/formatter, calendar handler, forms manager, interoffice correspondence package), and software development tools (including a requirements tracing tool, SREM, also part of the SDS environment discussed in Section 2.4, program design language, and a Fortran-77 analyzer).

UNIX[3]
: [Kern81] is sometimes referred to as a programming environment, but does not fit well into that category because of its generality. UNIX contains a multitude of basic tools (well over a hundred) that in general are not oriented to any specific programming language or to any specific life cycle phase. Consequently, UNIX is often used as a building block to build more specific environments. This aspect of UNIX is discussed further in Section 3.1.

Platine
: [Metz83] is an environment that consists of a methodology (the Platine methodology) and a set of tools (the Platine tools). The methodology consists of defining a software structure hierarchy, which produces typed abstract objects which are then associated with one of the following elements: source, listing, binary, map, nomenclature, or status. The methodology also includes the

[3]UNIX is a trademark of AT&T Bell Laboratories

production of software (merging of the elements), project management, and evolution. The environment includes LSTR (for the specification of real-time embedded systems), SDL (for system design representation), Metacomp (a compiler generator), EPCS (a project management tool), a formatter, a screen editor, a documenter, a mail system, crossrf (a data dictionary cross referencer), complex (a complexity measure), a configuration controller, and a comparator.

From the above examples, one can see that the emphasis of a general environment is to provide the user with a toolbox that can be applied to all phases of software development.

3.0 INTEGRATION

When an environment closely unites its major functions or activities, it is considered to be integrated. An interface carries information between the user and the environment, or between the environment and the tools which it invokes. The level of integration increases according to the amount of information conveyed, and the services automatically performed when an interface is invoked. Since there are many interfaces in an environment, there are many ways to achieve integration. User acceptance of an environment usually requires that the user interface be integrated; that is, the environment keeps track of what the user does and provides appropriate services. For an environment to perform efficiently on a computer system, the interface to the machine must be integrated. Environments contain tools that often communicate with other tools. This communication is sometimes facilitated through an integrated database interface.

The following sections discuss a more orderly way in which integration may be viewed, i.e. "levels" of integration. The two most important interfaces in an environment, the user interface and the database interface, are discussed. Finally, some of the issues associated with "loose integration" and "tight integration" are presented.

3.1 Levels of Integration

A way in which to view the interfaces in an environment is to model a system as a series of abstract levels similar to the model shown in Figure 3.1. The users are at the top level. Each lower level has features and characteristics that are less powerful in terms of software engineering. The lowest level is the machine, i.e., the computer hardware itself. The intermediate levels of the model are the software engineering environment.

Without an environment, the machine and the user interface would be the same, resulting in the user interface that early computers had in the 1940's and that a few "bare-bones" micros have today.

The requirements for the Ada Programming Support Environment (APSE) [Buxt80] defines a hierarchical model with levels of integration. The hierarchical model is a four level model:

level 3 - (Full APSE), a set of tools for full Ada programming support (life cycle, documentation, and management)

level 2 - (Minimal APSE), minimal tool set (editor, translator, linker, debugger, configuration manager)

level 1 - (Kernel APSE), supports database interactions, communications, and run-time

level 0 - the host level

Each level is viewed as a ring as shown in Figure 3.2. Users communicate primarily with the level-3 tools. Tools in each lower level communicate primarily with tools in the level immediately above or immediately below.

Some of the implementations of level-2 APSE's either developed or undergoing development

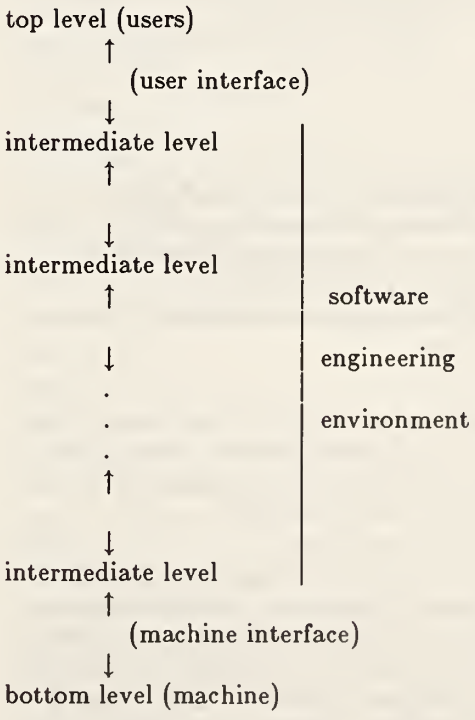

Figure 3.1 Software Engineering Model

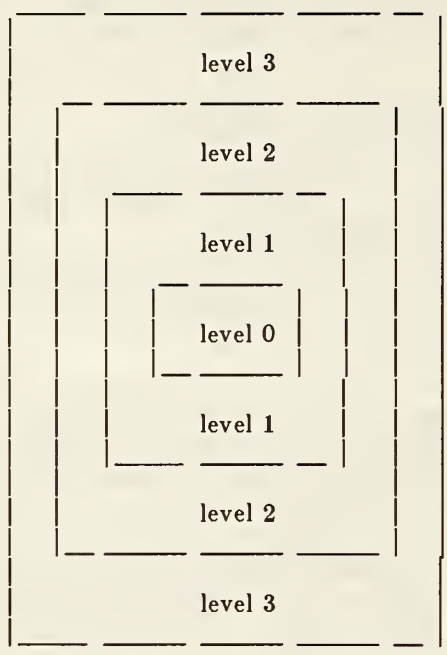

Figure 3.2 Hierarchical Model of the APSE

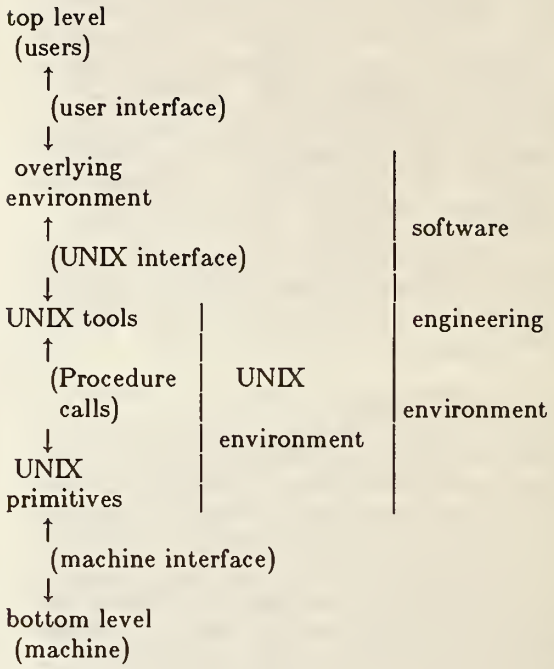

Figure 3.3 UNIX as an Underlying Environment

include:

- The Data General Corporation/Rolm Ada Development Environment (ADE)
- Ada Integrated Environment [Inter82]
- Ada Language System [Wolf81]
- Olivetti/Danish Datamatics Centre/Christian Rovsing Ltd.

It is expected that these and other future Ada environments will eventually include level-3 capabilities [Tayl84]. There is also a proposed standard interface to level-1 [CAIS83].

Many environments use existing systems as an intermediate level. The system most often chosen is the UNIX Environment. Examples of environments that have UNIX at a lower level include SPS, ARGUS, Toolpack [Cowe83], USE [Wass83], and Joseph [Ridd83]. Figure 3.3 depicts an environment containing UNIX.

The user communicates directly with the overlying environment and may even be unaware that UNIX is at a lower level. However, most overlying environments provide an ability to communicate directly (an escape) to UNIX. The overlying environment uses the UNIX interface to invoke UNIX tools and the UNIX tools communicate with the underlying UNIX primitives.

UNIX is chosen as an underlying environment for several reasons:

(1) UNIX Primitives. UNIX is available on many machines because the UNIX primitives are small in number and easy to define. The only major obstacle to portability is the availability of a C compiler because most of the UNIX tools are written in C. Since the overlying environment relies on UNIX, it can be made available on all the machines that UNIX runs on (within memory and disc limitations, of course).

(2) UNIX tools. UNIX contains over 100 tools. These tools can be used as building blocks to the overlying environment, thus eliminating a lot of development of basic underlying environment functions.

(3) UNIX Interface. The UNIX interface is called the shell. The shell is in many respects a very high level language (VHLL),

because it allows the user to use tools within the environment as objects. This is accomplished in a manner similar to a programmer's use of variables in a high level language. The output from one tool can be directed (piped) as input to another tool. It is this capability that makes the underlying UNIX tools suitable for building blocks to an overlying environment. In fact, one could argue that UNIX is more suited for building than it is for direct use due to the resulting "unfriendliness" of the interface. This issue is discussed further in the next section.

(4) UNIX File System. Because UNIX files are defined as strings of characters, the UNIX file system can be used as an underlying database. The shell allows the file system to be defined hierarchically and the files can be easily manipulated. Consequently, the products of software engineering can easily be tagged, stored, and retrieved from the file system in a manner similar to a hierarchical database.

UNIX is not the only environment that is chosen as an underlying environment. Many environments rely on operating systems, for example Platine [Metz83] incorporates text processing and editing tools from Digital Equipment Corporation's VAX/VMS. An interesting environment that is based on the book, "Software Tools" by Kernighan & Plauger [Kern76], is called the Software Tools Virtual Operating System (STVOS) [Hall80]. STVOS has a structure similar to UNIX, except that most of its tools are written in a portable subset of Fortran. Since Fortran is highly portable, STVOS has a long list of machines and operating systems that it runs with.

3.2 User Interface

As shown in Figure 3.1, the user interface is the interface between the user and the software engineering environment. One way that a user perceives an integrated interface is by how well the system is human engineered. Many of the issues associated with human engineering are addressed in Section 4. However, the most important issue associated with the user interface is primarily an interaction issue. That is, does the environment keep track of what a user is doing and does it provide services that are in context with the user's current state? In many respects, this issue is a research issue in artificial intelligence and expert systems, but current environments can address this issue to a limited extent. Section 2.3 addressed how programming environments can provide incremental development services that are in context with a programming language. However, a more basic way to provide this service is to have a menu interface.

Historically, command interfaces have been used for software systems because they are simpler to implement and easy to update and modify. Menu interfaces have been avoided because they often are too slow to appear on the screen. A user's performance and attention span are easily affected while waiting for the menu to be placed on the screen. These issues are implementation issues and do not necessarily apply to today's technology. Intelligent terminals, user workstations, and single user microcomputer systems make menu systems easier to implement and can provide a large flow of information between an environment and a user. In fact, a bit-mapped terminal allows a menu to "pop" on the screen with almost no perceived delay.

If a menu interface is well engineered, a user perceives a well integrated system because he or she has available all options needed at any usage state and no more. This greatly reduces the amount of conditioning required for a user to become familiar with a system. It can also put the user in the right frame of mind for the task that is being accomplished.

An example of a software engineering environment that has a well-engineered user interface is Argus. Figure 3.4 shows the menu that appears on the screen when a user first initiates the system. The user has several options available. The menu choices at the top of the screen never change and are always available. The menu options in the middle of the screen change depending upon what state the user is in. If software is being designed, then the user would select the designer's toolbox. This selection causes a menu in which only design options are available, the system helps the user get into a designing frame of mind by not distracting the user with options that are available in the other toolboxes.

UNIX is an example of an environment that is command oriented. It was mentioned previously that UNIX has over 100 tools available to the user. Each tool is invoked with a different command. Most of these commands have a series of parameters that are associated with them. The combined functionality of these tools makes UNIX a very powerful environment, but unfortunately most users of UNIX tap only a small part of this power. The reason is that users tend to learn a minimal subset of UNIX, a subset that will just barely solve most of their problems. The fact that another tool may solve a problem better may never be discovered by the user. Consequently, a lot of UNIX's power goes virtually unused.

The previous section mentioned that UNIX underlies many environments. Many of these environments help to harness some of that unused power

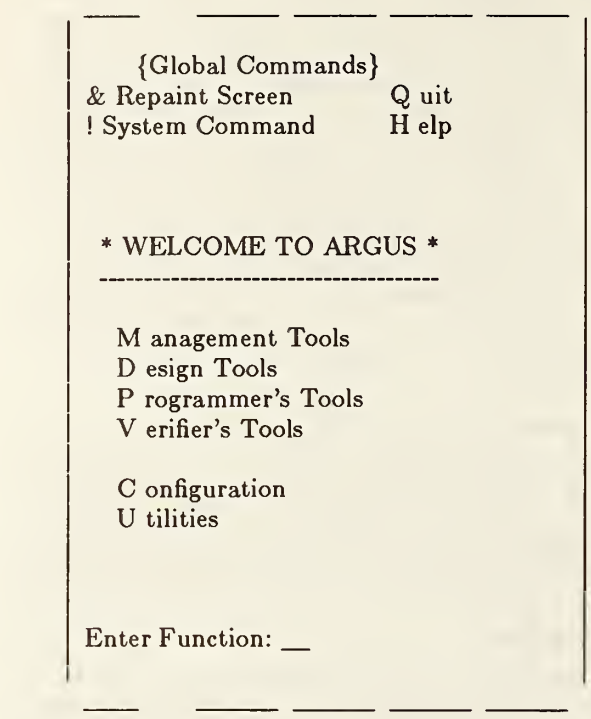

Figure 3.4 The Top Level ARGUS Menu

of UNIX by specifically choosing features that are oriented to software engineering applications. It is likely that some of the features chosen would never be used by the average software engineer.

3.3 Database Interface

Tools may exist at the same level in the environment or they may be at different levels. This section concentrates on the interface between tools and the database. The database, in this context, is the repository where a tool deposits its output or gets its input, unless of course, there is another interface with the database. For example, if an editor is part of the environment, it will likely receive input from both the user and the database, but its output will definitely go to the database.

The products stored in a database can include requirements and design specifications, high and low level programs (load modules, assembly programs, etc.), test input and output, documentation, etc. Associated with each item in the database may be a number of attributes such as the type of an item, stage of development, time of last update, access rights, author, version number, length of item, disk or memory address of item, etc. The interface to the database can be environment specific or it can be implemented by an available, possibly commercial, database management system, file system, or library system.

The database in CDL2 [Baye81] is environment specific and central to the environment. The information stored in the database forms a hierarchy according to authorship and language constructs. The top levels provide a hierarchy by module ownership; the bottom levels correspond to some construct of the programming language.

Another example of an environment specific database is the one that is used in FASP [Steu84]. FASP is a large scale environment that is hosted on five mainframe Control Data Corporation (CDC) computers. The database in FASP consists of the following libraries:

source library
object library
test library
interface data library
production data library
documentation library

The interface data library contains information such as linkages to external object programs or to shared source code. The production data library contains modification histories and other management information. FASP manages these libraries as a whole rather than as distinct parts. For example, there is one-to-one correspondence between the products in the object library and the products in the source library.

SPS [Boeh84], on the other hand, has a master project database that includes the UNIX hierarchical file system, the UNIX source code control system (SCCS), the Ingres relational DBMS, and the IDM-500 database machine. The UNIX file system is used to store the products of software development. SCCS is used to track successive updates of each product. Ingres and IDM-500 are used to keep track of the relationships between the products. Because SPS uses four independent tools with four distinct interfaces, some integration of the interfaces has been done. The integration of these tools is an example of the building block capability that was previously mentioned as an advantage of using UNIX as an underlying environment.

3.4 Binding the Tools in an Environment

In a sense, the database is a tool that is used by other tools and therefore is the interface between two tools. This section generally discusses the interface among all tools in the environment.

The interface in an environment can be loosely or tightly bound to, or integrated with, a software development methodology. As mentioned previously, most framing environments are oriented to a development methodology. Therefore, one can expect that these environments are tightly bound to a methodology. The outputs of a tool are expected as input by the next tool in a format consistent with a given methodology. If an interface is tightly bound to a software development methodology then the functions provided by the environment are tightly integrated. There is an order to the environment that should be followed and may even be forced by the tools because the input to one tool is highly dependent on the output from another tool.

Tight integration has the advantage that it encourages the correct use of a methodology. It can greatly reduce the number of management controls that are necessary to ensure that all the various software products are developed and, in some cases, even check their correctness. The disadvantage is that a tightly integrated environment lacks flexibility. If a user wants to introduce a new methodology or a change to the current methodology, the environment may not be flexible enough to support it.

If an interface is loosely bound to a software development methodology then the functions provided by the environment are loosely integrated. The tools in the environment are not dependent on one another and no order is expected. Most general environments are loosely integrated. As one would expect, the advantages and disadvantages are opposite to those of tight integration. Loose integration is methodology independent but requires management control to ensure a methodology.

Programming environments may be loosely or tightly integrated. In particular, if an environment is highly language dependent then it is probably tightly integrated. For example, tools in the environment may be highly dependent on the products produced by a syntax analyzer or graph generator.

Examples of tightly integrated environments include CDL2 [Baye81], POE [Fisc84], Interlisp [Teit81], CPS [Teit81a], and USE [Wass83]. Examples of loosely integrated environments include SPS [Boeh84], Toolpack [Cowe83], STVOS [Hall80], UNIX [Kern81], and ARGUS [Stuc83].

4.0 HUMAN FACTORS

As the advantages of software engineering environments are realized and the demand for environments increases, the quality of an environment will become an increasingly important issue. From a user viewpoint, quality is often measured in terms of the human factors that are addressed by the environment. If it is difficult to understand how to use an environment or if becoming an expert user requires a lot of conditioning, it is likely that an environment will not be widely used.

In the past, human factors were not always considered an important issue for computer operating systems. Many systems that measure poorly in terms of human factors, including most traditional operating systems, have withstood the test of time. It is important however, to distinguish who the real users of these systems are. The real users are the systems programmers or system "gurus". They are the experts on how to best use the capabilities of a system and they are, for many applications, the interface between the user and the system.

Software engineering environments, on the other hand, should not require a system expert as an interface between the software engineer and the environment. The operating system, either alone or via the software engineering environment, should be naturally oriented to software engineering; the user should be able to make full use of the operating system without unnecessary conditioning. In other words, the software engineer should only need minimal exposure to the operating system to be productive and should not have to get used to special system modes, exceptions, quirks, bugs, etc.

This section discusses some of the factors that reduce the amount of conditioning that is necessary to make full use of a system, whether an operating system or a full software engineering environment. In particular, this section presents ways to provide on-line assistance; some quality factors; and finally, implementation techniques.

4.1 On-Line Assistance

Since the early days of the University of Illinois' PLATO educational system, computer systems have provided on-line assistance to the point where it is now offered on most mainframe systems, many minicomputer systems, and even a few microcomputer systems. Users of computer systems have become accustomed to the convenience of on-line assistance.

On-line systems provide a wide range of assistance to users varying from simple command assistance to elaborate and detailed tutoring [Houg84]. The sections that follow discuss and exemplify the types of assistance provided by systems.

4.1.1 Command Assistance

Command assistance is the most common type of assistance provided by on-line systems. This assistance can be obtained through various access

methods such as the issuance of a HELP command, pressing an explanation key, or typing a question mark [Rell81]. In each of these cases, a command must be specified along with the request for help. For example, on the Digital Equipment Corporation's VAX/VMS Operating System, if a user wants print assistance, then the command HELP is typed followed by PRINT. Figure 4.1 shows the result of typing HELP PRINT on the VAX. The example shows that a brief explanation of the print command is presented followed by a list of possible parameters that can be specified with the print command. Note that more detailed assistance is also available on each of the possible parameters.

4.1.2 HELP Assistance

Command assistance is valuable only if a user knows the name of the command on which assistance is needed. For example, it may be that a user is unsure whether assistance is needed for the commands: PRINT, WRITE, TYPE, LIST, or DUMP. Rather than guess at commands, many systems provide a way of determining the commands that are available on the system. For example, on many systems the user may type HELP HELP (or just HELP) to determine the type of assistance that is provided on the system.

```
HELP PRINT

PRINT
 Queues one or more files for printing,
 either on a default system printer or
 on a specified device.

   Format:
     PRINT file-spec[,...]

 Additional information available:
   Parameter Qualifiers:
     /AFTER=absolute-time
     /DEVICE=device-name[:]
     /NOHOLD (D)
         .
         .
         .
```

Figure 4.1 Print help on the VAX/VMS

4.1.3 Error Assistance

Error messages issued by a system are usually very brief and frequently need further clarification by a user. There are systems that provide further on-line assistance for error messages. For example, a user may request the meaning of an error by typing "HELP ERROR <code>", where <code> is some code identifier for the error that occurred.

4.1.4 On-line Tutor

One of the most difficult problems for a new user of a system is simply getting started. Lists of commands and command descriptions do not help the new user because for many systems the amount and depth of the information is too advanced for the beginner. A new user needs a step-by-step tutorial introduction with exercises and a capability to try out various commands without doing harm to oneself or any other users on the system. An example of a system that provides this capability is the SIGMA message processing service [Roth79]. This system includes a tutor with on-line lessons, on-line exercises, a protected mode for exercises, and a toggle which allows the user to switch back and forth from the tutorial to the exercise mock-up of the system.

4.1.5 On-line Documentation

Another approach to on-line assistance is to make traditional user documentation available on-line. With retrieval mechanisms, this type of assistance has the advantage of ensuring assistance and user documentation are one and the same [Pric81]. An example of a system that provides this capability is UNIX. UNIX provides a command called "man" (short for manual) that retrieves from one to several pages of manual documentation (the editor documentation is 7 pages). Figure 4.2 is an example of the documentation provided for the UNIX "cmp" command [UNIX42].

One problem with on-line documentation is that many times the depth of the information provided is invariant. This means that a more experienced user who is only interested in a specific aspect of a command must wade through all the documentation to get the information desired. This problem can be alleviated somewhat by allowing more sophisticated retrievals on a command. An example of a system that provides this capability is IBM's Time Sharing Option (TSO). The TSO allows the user to enter a keyword (such as "SYNTAX") that narrows the scope of the documentation that is provided.

4.1.6 Other Types of User Assistance

Many other types of user assistance are described by Relles [Rell81a]. They include:

CMP(1) UNIX Programmer's Manual CMP(1)

NAME
cmp - compare two files

SYNOPSIS
cmp [-l] [-s] file1 file2

DESCRIPTION
The two files are compared. (If file1 is '-', the standard input is used.) Under default options, cmp makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

-s Print nothing for differing files; return codes only.

SEE ALSO
diff(1), comm(1)

DIAGNOSTICS
Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

Figure 4.2 On-Line Documentation for the UNIX "cmp" Command

Question or Prompt Assistance - explanation of a displayed question or prompt.

Example Presentation - Presentation of an example of a correct or valid input.

Definition Assistance - Explanation or definition of a term.

Menu Assistance - Presentation of a list of allowable commands.

System Status Displays - Presentation of various system parameters, such as swapping ratios or system performance metrics.

News - Displays of news that is of interest to system users.

4.2 Quality Factors

In each case, the information presented to the user must be developed properly or the purpose for providing the information will not be achieved. Similarly, any of the features of a software engineering environment, should be directly applicable to the user's task and provide minimal distraction. The following sections discuss some of the quality factors that should be addressed in the development of an environment.

4.2.1 Robustness

Fenchel [Fenc81] provides an important list of quality attributes that are necessary for a system to be robust. These attributes promote productive use and user acceptance of systems. They include:

Availability - To promote user confidence in a system, it should always be accessible at any point during user interaction. It should never leave a user in an uninterruptable or dead state.

Accuracy - The information provided to a user must be accurate and up-to-date. Inaccurate information will cause a user to lose some confidence and trust in a system. Repeated inaccuracies will cause a user to lose all trust in a system.

Consistency - It is important that information have a consistent presentation, reading level, scope, and length. It is unreasonable to expect users to tolerate various presentation formats, the use of sophisticated terminology or "buzz-words", varying scope of

coverage from one request to another, or overwhelming detail for certain assistance requests.

Completeness - All details should be covered. A user should not be surprised by a system. All system features, commands, parameters, and consequences of these should be covered and explained.

4.2.2 Pitfalls

A paper by I. A. Clark [Clar81] discusses several pitfalls that one should avoid in the development of a system. Some of the pitfalls are very specific to the simulation environment discussed in the paper, but many are generalizable and include the following:

Confronting a User with a Solid Block of Text - Being confronted with a solid block of text is neither helpful nor reassuring to a user. Information presented to a user should be stylized so that a user can recognize at a glance the significance of the format and grouping of words. It is better to start each sentence on a new line than to present a user with a paragraph of right and left justified text.

Pitching the Reading Level too High - Pitching the reading level too high tends to distract the user from the real problem that the system is being used for. Clark recommends the use of the "fog index", a measure of readability, [Gunn59] as a rough and ready yardstick for measuring the reading level. A recent study by Roemer and Chapanis [Roem82] supports this position. They found by experimentation that regardless of ability, subjects preferred the 5th grade reading level of a tutorial where the study involved 5th, 10th, and 15th grade reading levels. They concluded that the most sensible approach in designing computer dialogues is to use the simplest language, especially when the dialogue involves instruction or procedures. Also, well-designed, human-engineered dialogues can greatly improve users' attitudes toward computers.

Timing Instructions to the User - Users prefer concise and direct instructions. If a user is told to perform an action, it is likely the user will perform it. This can cause problems if the system is in some special mode, such as a tutorial or instructional mode, and is not prepared to perform the directed action.

4.2.3 Other Quality Factors

Although Clark and Fenchel cover many of the quality factors, there are two outstanding factors that should be mentioned [Shne80] [Rell81b].

Communication Overload - When a system communicates with a user, the communication should be brief, direct, and concise. If it is not, then it is likely that the users will be overloaded with too much information. Communications should follow the "seven plus or minus two" rule [Mill56]. That is, they should not introduce, in general, more than seven concepts in a single frame. If it is necessary to provide more detail, then the information should be structured using the query-in-depth design technique that is discussed in the next section.

Anthropomorphization - Communications should not be anthropomorphized. For example, the message

> When you type the command "Logout", I will abort your job

contains two anthropomorphic references. The use of "I" and "abort" ascribe human attributes and conduct to a computer operating system. References such as these cause users to view such a system as something other than the tool it really is. A better message would be

> Logout terminates a session.

Another anthropomorphic issue that should be avoided in communications is placing blame on the user. Messages should reflect system limitations and not a user's inadequacy. For example, a misdial on the telephone often yields the following communication from the phone company:

> We are sorry, but the number you have dialed...

In this message, the phone company is apologizing for its inability to make sense out of the number dialed, even though it is likely that the dialer is at fault.

4.3 Implementation Techniques

This section presents some of the techniques that can be used in the development of systems that incorporate human factors.

4.3.1 Query-in-Depth

Levels of help or query-in-depth is a popular issue in the implementation of systems. It is mentioned in several references [Rell81] [Fenc81] [Roth79]. One would expect that many future systems will implement this capability. The advantage that

this capability provides is that a user can obtain successively more detailed assistance if it is needed simply by asking for it. Relles and Price [Rell81b] demonstrated a system with this capability at the 5th International Conference on Software Engineering. In their system, the user requests assistance by typing "?". Further use of "?" provides the user with more in-depth explanations until the user reaches the lowest level, at which point, a person and a phone number is provided.

4.3.2 Contextual Assistance

Assistance that is context sensitive is another popular technique. Unlike query-in-depth, contextual assistance can be found in many systems today. For example, the Digital Equipment Corporation's TOPS-20 operating system will on request complete the name of a command or file if enough characters have been typed to make the request non-ambiguous. On the other hand, if the request is still ambiguous (not enough characters have been entered) then a user can request a display of the available options. This means that the system must be able to determine at any point in the user-machine dialogue a menu of choices that are available.

Query-in-depth and contextual assistance are capabilities that can provide a very powerful help system when they are used together. In the previous example[Rell81b], notice that both capabilities are provided. If the system had not kept track of the user's entries then the second request for assistance would have produced the same message as the first.

Fenchel and Estrin [Fenc82] have furthered this technique whereby assistance is integrated into the grammar that is processed by a combined parser generator and an integral help generator. This work has been integrated into the SARA support environment. With this approach, the integral help generator has a concise representation of the user interface available to it, so contextual assistance is easily generated.

4.3.3 Natural Language

Communication by natural language is limited by a system's ability to make inferences. When a statement cannot be logically resolved, inferences must be made. For example, the following request will require an inference.

> Find all processes that communicate with proc1.
>
> List the processes that begin with the letter "T".

Is the second request related to the first request or not? In this case, it is obvious that it is. Why process the first request if it is not. An inference would not have to be made if the second request was phrased as follows.

> List only those processes that begin with the letter "T".

The ability to make inferences requires knowledge of the task being performed and of the person performing the task. If inferences are not made, then clarifications must be requested by the system. Users may prefer to deal with a concise command language or selections from menus than to "train" a natural language interface. It is therefore especially important to test thoroughly natural language communications to insure user acceptance before releasing a system.

4.3.4 Simulation and Prototyping

Clark [Clar81] discusses the use of software simulation in product development. In particular, a prototype of a system was developed that simulated the user interface. Many problems, including those discussed in Section 4.2.2, were discovered early in the development of the product. Since errors or changes are much more costly in the later stages of development, this is a cost-savings technique.

Wasserman and Shewmake [Wass82] have also reported success with a similar technique for the development of information systems. They have a development methodology and a supporting environment built around the development of prototypes. The methodology includes the following steps:

> Requirements Analysis (activity and data modeling)
>
> External Design (interface design)
>
> Creation of a "Facade" (prototype of user interaction)
>
> Informal Specification of the System Operation
>
> Database Design (relational)
>
> Creation of a System Prototype (some or all functions)
>
> System Design
>
> Implementation
>
> Testing and/or Verification

The environment includes tools for constructing prototypes. Some of these tools feature screen-oriented editing and browsing, and software management and control.

There are also several production environments that can be used to develop prototypes. Several of these were demonstrated at the first SoftFair Conference and include the following: CAP [Bass83], Proto-Cycling [Zajo83], Index [Rube83],

and PRIDE [Bryc83].

4.3.5 Contextual Mode Switching

When a person uses a software engineering environment, the user will be in one of three modes: using mode, learning mode, or software engineering mode. A user is in "using mode" when the thought process is focused on how to use the environment to perform the software engineering task. For example, if a user is editing a document,"using mode" would focus the thought process on the command to insert or delete words, lines, etc., instead of the material to be edited. As a user becomes proficient with an environment, "using mode" can almost become subconscious, like a typist using a typewriter.

A user is in "learning mode" when the thought process is focused on acquiring skills for the "using mode". The rest of the time the user is in "software engineering mode", that is, the user's thought process is on developing software. Contextual mode switching occurs when a user goes from one mode to another. For example, if a user does not know how to use an environment feature then he or she goes from "software engineering mode" to "using mode" to "learning mode" back to "using mode" and finally back to "software engineering mode". Mode changing can significantly decrease performance. For example, Clark [Clar81] found that in some cases mode switching can cause users to forget why they went into "learning mode" in the first place.

One way to make mode changing less disruptive is to place using and learning information on the screen simultaneously with the problem. A system that provides this windowing capability is the Automated Interactive Simulation Modeling (AISIM) system [Aust82]. For example, when a designer is editing a flow-diagram, the command "MENU" can be issued to have a menu drawn to the left of the diagram. With both the menu and the diagram on the screen, there is not as much disruption as there would be if the diagram was replaced by the menu. Other systems that provide a similar capability include Cedar [Teit84] and Apple's Lisa and MacIntosh.

Another way to ease mode switching is to provide contextual assistance as described in Section 4.3.2.

5.0 ANALYSIS AND SOFTWARE QUALITY

In this section, the taxonomy developed for FIPS PUB 99 [FIPS99] is used as a basis for presenting various analysis and quality features. Whereas the taxonomy is concerned with a broad range of features offered by all types of software development tools, the scope of this section will narrow on static analysis, dynamic analysis, and management features that are appropriate for software analysis and quality assurance.

In a software engineering environment, analysis should be provided to whatever extent is possible. For example, if the environment supports the development of requirements and design specifications, then those analysis features that are appropriate for these specifications should be provided. This is particularly important since detecting errors as early as possible in the life cycle is a proven cost savings technique. In this case, the environment should provide the analysis even if the results are incomplete. For example, it may not be possible to audit software completely until all the code is available and compiled. In this case, the auditing feature should provide as much detail as possible pertaining to what is available.

5.1 Static Analysis

Static analysis features specify operations on the subject without regard to its executability [Howd78]. The subject can be a specification language, a high level language, or even documentation. Static analysis features describe the manner in which the subject is analyzed.

Auditing
: Conducting an examination to determine whether or not predefined rules have been followed. Examining source code to determine whether or not certain standards are complied with is an example of this feature. Auditing could include such checks as finding missing parts of a system and identification of poor and dangerous software engineering practices.

Completeness and Consistency Checking
: Assessing whether or not an entity has all its parts present and if those parts are fully developed and externally and internally consistent [Boeh78]. In order to test automatically for completeness and consistency of requirements and designs, a formal specification language is required. The environments associated with PSL/PSA [Teic77] and SREM [Alfo81] are examples that provide these capabilities. Environments which use formal proofs to demonstrate consistency of specifications and code have also been developed for critical applications such as computer security. Examples include AFFIRM [Thom81] and Gypsy [Ambl77].

Complexity Measurement
: Determining how complicated an entity (e.g., routine, program, system, etc.) is by evaluating some number of associated characteristics [McCa76] [Hals77]. The following characteristics can impact complexity: instruction mix, data references, structure

and control flow, number of interactions and interconnections, size, and number of computations.

Cross Reference
: Referencing entities to other entities by logical means. Although cross references are easily generated from symbol tables, they are not often used because of the massive amount of information they contain. Even cross reference listings of small programs are quite lengthy. This problem can be alleviated somewhat by retrieval mechanisms associated with a management feature called a data dictionary. This feature is presented in Section 5.3.

Interface and Type Analysis
: Checking the interfaces between system elements for consistency and evaluating whether or not the domain of values attributed to an entity are properly and consistently defined. Environments that support specification languages and strongly-typed high level languages such as Ada and Pascal almost always provide these features. However some of these languages support separate compilation of procedures which can delay interface checking until the procedures are linked. This is not a recommended practice because it uncovers errors very late in the development process. With knowledge of the design and data specifications, both analyses can be performed earlier in the development process.

Scanning
: Examining an entity sequentially to identify key areas or structure. Examining source code and extracting key information for generating documentation is an example of this feature.

Statistical Profiling
: Performing statistical data collection and analysis of statement types. Statistical profiling can provide useful information to language designers and managers. Managers can use this information to determine programming style among their staff. For example, the data could show that a programmer avoids certain advanced constructs or overuses poor constructs such as GO TO statements. Language designers and standards committees find statistical profiles useful in identifying popular and unpopular language constructs. The usefulness of statistical profiles is emphasized in a classic paper by Knuth [Knut71].

Structure Checking
: Detecting structural flaws within a program. Structure checking is a common feature offered by many compilers. Results of a survey of compilers [NBS418] found occurrences of the following types of structure checking in Fortran and Cobol compilers: unreachable statements, null-transfer statements (i.e., a branch to the next statement), null-body loops, empty programs, and self-transfer statements. In addition to these types of structure checking, there are also tools that check for violation of structured programming constructs (e.g., COBOL STRUCT [FSWE80]), and in some cases restructure code (e.g., The Engine [Lyon81]).

I/O Specification Analysis
: Analyzing the input and output specifications in a program usually for the generation of test data. Analyzing the types and ranges of data that are defined in an input file specification in order to generate an input test file is an example of this feature.

Reference Analysis
: Detecting errors in the definition and use of data. To check completely for this type of error, it is necessary to generate a program graph and to check data references on each path. With this level checking, errors such as variables defined but never used, variables used but never defined, and variables defined and then subsequently redefined before being used can be checked on all paths through a program. The tool which promulgated this technique is DAVE [Oste76] which has been made part of the analysis capability of Toolpack [Cowe83].

5.2 Dynamic Analysis

Dynamic analysis features specify operations that are determined during or after execution takes place [Howd78a]. Dynamic analysis features differ from those classified as static by virtue of the fact that they require some form of symbolic or machine execution. Dynamic analysis is the technique used to derive meaningful information about a program or system's execution behavior.

Assertion Checking
: Checking of user-embedded statements that assert relationships between elements of a program. An assertion is a logical expression that specifies a condition or relation among program variables. Checking may be performed with symbolic or run-time data. Assertions, which can be implemented as special comments, are useful as an understanding mechanism. Assertions can be used to declare relationships or states that are assumed to be true at certain points in a program. Thus, a programmer can use

assertions for debugging. Assertions can also represent relations or states assumed true at a higher level of abstraction. Using assertions in this manner allows one to test consistency between a design specification and the code. If assertions are placed in the code so that there is a mapping from each assertion to each design or requirements specification, then assertions can be used to compute design or requirements coverage. Assertion checking is provided by RXVP80 [Saib81] and the NBS Fortran-77 Analyzer [NBS359].

Coverage Analysis

Determining and assessing measures associated with the invocation of program structural elements to determine the adequacy of a test run [Fair78]. Coverage analysis is useful when attempting to execute each statement, branch, path, or iterative structure in a program. Since coverage analysis yields actual testing metrics, its most important impact is that it encourages programmers to develop testable programs. That is, in order to get a high percentage of program coverage, a programmer must try to make all parts of the program accessible for testing. NBS Special Publication 500-88 [NBS88] lists 40 tools that provide this feature.

Regression Testing

Rerunning test cases which a program has previously executed correctly in order to detect errors spawned by changes or corrections made during development or maintenance. Environments provide regression testing through a capability to automatically "drive" the execution of a program through its input test data and report discrepancies between the current and prior output.

Simulation

Representing certain features of the behavior of a physical or abstract system. Simulation of the user interface is a capability that is built into some software engineering environments, such as USE [Wass83].

Timing

Reporting actual CPU, wall-clock, or other times associated with parts of a program.

Tracing/Debugging

Tracking or monitoring the historical record of execution of a program either to increase understanding of its behavior or to detect and correct errors. Environments can offer many tracing and debugging features including: breakpoint control, data flow tracing, and path flow tracing.

5.3 Management

Management features aid the management or control of software development. There are three major types of management features.

Configuration Control

Aiding the establishment of baselines for configuration items, the control of changes to these baselines, and the control of releases to the operational environment.

Project Management

Aiding the management of a software development project. Where configuration control aids all participants in the software development process, project management features primarily aid the project manager. Common project management features include cost estimation, resource estimation, and scheduling. Cost estimation has been the subject of concentrated research which has resulted in many cost models [DACS79]. TRW's SPS incorporates the COCOMO cost model [Boeh81]. Scheduling is closely associated with both cost and resource estimation and deals with software engineering personnel and activities. Environments with these project management features commonly provide milestone charts, personnel schedules, and activity diagrams as output.

Information Management

Aiding the organization, processing, accessibility, modification, and dissemination of information that is associated with the development of a software system.

There is much information associated with the development of a software system. Since much of the information requires special features in an environment to handle the different types of data, information management is further subdivided.

Data Dictionary Management

Aiding the development and control of a list of the names, lengths, representations, and definitions of all data elements used in a software system. Data dictionary systems have been available for years [NBS3] but their potential in a software engineering environment has not been fully realized. Data dictionary systems can be a valuable tool over the entire life cycle of systems and software [FIPS76]. For example during the requirements definition phase, information is collected about data type and usage requirements. This information can then be used in the production of data descriptions for individual software packages and procedures, and in conjunction with other analysis capabilities (such as cross reference and reference analysis).

Documentation Management
: Aiding the development and control of software documentation. Many of the static and dynamic analysis features previously mentioned automatically develop documentation that if properly used can significantly reduce documentation costs.

Code Management
: Providing and controlling access to files containing programs or parts of programs. Code management is essential for large software development. The UNIX environment includes a tool called "make" [Feld79] which effectively manages code, but can be extended to other software products such as specifications and documentation. Make reads a specification of the structure of a program and puts together an up-to-date version of the program based on its specification.

  Code management can also provide a way to avoid redundant coding by identifying the functionality of programs or program fragments. Code management can keep track of the functionality and retrieve information about programs that provide capabilities of interest to the software engineer. Although capturing the functionality of a program fragment is a current subject of research, an interim solution can be implemented through keyword specification and retrieval in a manner similar to library systems. A recent paper [Lecl82] describes a browse documentation system that is similar in concept.

Specification Management
: Aiding the development and control of requirements and design specifications. Specification management is somewhat methodology dependent because usually there are formal procedures for the definition and use of the specifications. However, the environment may or may not include controlling elements such as review and approval mechanisms, required record keeping of scope changes, strategy changes, justification, and rectification.

Test Data Management
: Aiding the development and control of software test data. Test data management is necessary to provide the dynamic analysis feature of regression testing.

5.4 Underlying Analysis Features

Many of the features presented in Sections 5.1, 5.2, and 5.3 imply the existence of more primitive features in the environment. These underlying features could be thought of as an intermediate level in the model discussed in Section 3.1 and as shown in Figure 5.1.

A list of underlying analysis features includes the following.

data base management - support information management and project management features.

graph generation, graph analysis, and data flow analysis - support complexity measurement, structure checking, and reference analysis by constructing, traversing, and analyzing a specification of the program graph.

profile generation - supports complexity measurement and statistical profiling.

comparison - supports regression testing and configuration management.

lexical analysis, syntax analysis, semantic analysis, parsing, and symbol table generation - support most all of the analysis features that deal with programs.

instrumentation - supports assertion checking, coverage analysis, and timing analysis.

6.0 SUPPORT FOR DIFFERENT TYPES OF USERS

Software engineering may involve the participation of many people performing many different tasks. Division of labor and specialization of function is the means by which chaos is avoided in a software engineering project. In addition, a tree-like hierarchy diminishes the need for detailed communication among all the participants; for large projects, small teams should be at the lowest levels of the hierarchy. For small projects, one person may of course have responsibility for many different tasks.

Brooks [Broo75] defines two organizers in the team: a producer and a technical director. The role of the producer includes:

- assembling the team
- dividing the work
- establishing the schedule
- acquiring resources
- establishing the pattern of communication and reporting
- ensuring that team stays on schedule.

The role of the technical director includes:

- requirements analysis and specification
- developing high level designs
- specifying the interfaces
- sketching the internal structure

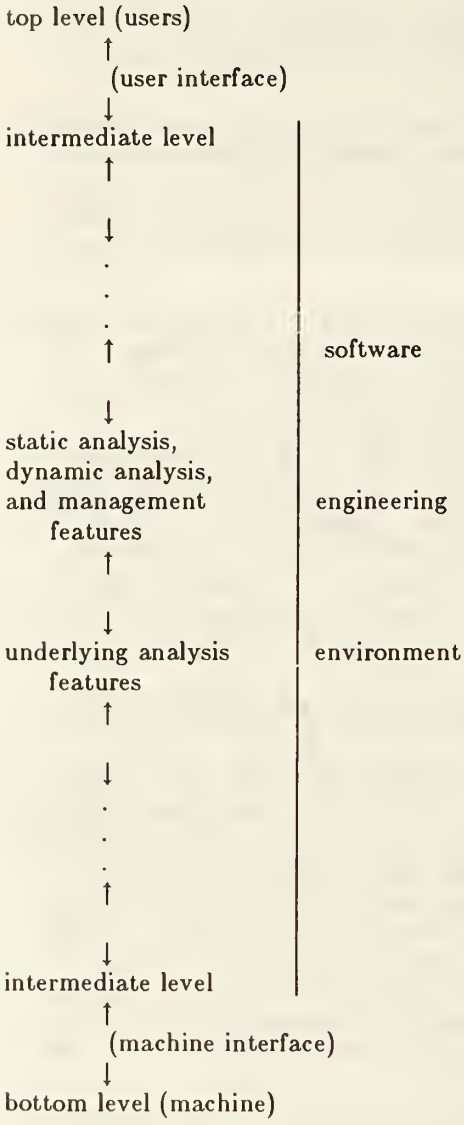

Figure 5.1 An Environment with Intermediate Levels of Analysis

- solving technical problems
- ensuring technical quality.

For small projects, the technical director and the producer may be the same person.

Other members of the team can be chosen from the following:

designer(s)
programmer(s)
analyst(s)
documentation editor
librarian
maintainer(s).

Once again, it is possible that in small teams the technical director may also be an analyst, a designer, and a constructor. It is also possible that other team members may also play different roles.

Regardless of the division of labor or the specialization of function, an environment should provide support for all the players. In particular, an environment should orient its support to the player that is currently using the system. For example, if the user is the documentation editor then office automation features should be emphasized and other features of the environment should be "hidden". Even if a user is wearing several "hats", such as the technical director who may at one time or another play all the roles, then the environment should emphasize the features that are associated with the hat that the user is currently wearing.

An example of an environment that explicitly divides its features into toolboxes that are oriented to the different players is ARGUS [Stuc83]. Figure 3.4 shows the top level menu for ARGUS which essentially asks the user to identify the role he or she is currently playing. Once the role has been identified, then the lower level menus include choices appropriate to the role. One particularly interesting aspect of ARGUS is the screen-oriented editor that is available in each of the toolboxes. Although this editor has a basic set of commands available in all toolboxes, special commands become available depending on which toolbox is currently active. For example, if the user is in the designer's toolbox then graphics commands are available so that the user can draw boxes, circles, arrows, etc. If the user is in the programmer's toolbox then commands that place language constructs, such as if-then-else or while-loop, on the screen are available for selectable programming languages.

The following sections discuss the duties of the various players and the features that they would expect to find in a software engineering environment.

6.1 The Manager

Brooks' producer [Broo75] is usually considered the manager of a software engineering project although it is quite likely that the technical director will occasionally get involved with this role. The manager acquires resources and allocates work according to the resources that are acquired. The resources can include people, hardware, software, and dollars. In addition, the manager worries about the "health" of the project, e.g.: Are people

happy with their current roles? Are they communicating? Are they on schedule? Does the software engineering environment give them the support they need?

In Section 5.3, features that aid the management and control of a software development project were presented. The features that are of particular interest to the manager are those described under project management. With the advent of microcomputers, many new tools have become available that are targeted for the manager and will be expected in software engineering environments. These tools include:

Electronic Spreadsheet - allows the user to retrieve information such as scheduling and cost data from the database, to manipulate this information, and to present it in column or graphic format.

Calendar Management System - allows the user to define and manipulate personal calendars, to set up reminders for future appointments, to print future schedules of appointments, travel, or other calendar events.

Telephone Book - allows the user to insert, delete, update, and retrieve phone numbers, may even perform auto-dialing.

Thought Organizer - assists the user in developing, organizing, manipulating, and reviewing facts, strategies, and concepts.

6.2 The Designer

It is generally recognized that a designer works at two different levels of design. Freeman [Free83a] defines the two levels as follows:

Architectural Design - determining the underlying structure of the problem from the requirements specification. When this structure becomes clear, an internal design of the system is devised. This design is necessarily at a gross level of detail. The parts of the system and their relationships, the basic algorithms that the system will use, and the major data representations and organizations that will be needed are all primary elements of the design at this stage.

Detailed Design - adding more detail to the major parts of the design. Precise algorithms and data structures are spelled out. Interfaces between parts are detailed. Hardware selections not made at the architectural level are made. Detailed design (as well as architectural design) may require several levels of refinement. This stage stops short of spelling out all programming details (e.g., housekeeping and local data structures).

In the early stages of design, it is quite likely that the technical director will be the only team member performing design. However, as the need for lower level designs begins to emerge, other members of the team may become involved with design.

Since an important goal of the designer is to document the system that is going to be implemented, techniques for communicating the design play an important role. Graphics, very high level languages, and text are the normal means for this communication. Environments should support all three, but at a minimum it should support text.

The two levels of design define two possible levels of support in the environment. Architectural design is supported by tools that assist the high level functional specification of the design. Examples include SARA [Estr78], DREAM [Ridd81], PSL/PSA [Teic77], AISIM [Aust82] and the tools associated with design methods such as the Parnas Method [Parn72], HIPO [Stay76], and Structured Design [Your75]. Architectural design is also supported by tools that support the high level specification of data flows within a system. Examples include the tools associated with Object-Oriented Design [Booc83] and the Warnier/Orr Method [Warn74].

Like architectural design, detailed design is supported by tools that assist the low level specification of functions and data. Tools associated with the many program design languages, such as Ada PDL [Priv82] or Caine, Farber, and Gordon's PDL [Cain75], assist low level specification of function. Tools associated with the Jackson Method [Jack75] or Object-Oriented Programming [Love83] assist the low level specification of data flow.

6.3 The Programmer

Of all the members of the software development team, the programmer is the member who traditionally receives the most support from software engineering environments. In Section 2 a programming environment was defined as one that concentrates on the activities that are performed during the latter part of the life cycle. These are the activities performed by the programmer.

The programmer is supported by features that are oriented toward a programming language with particular emphasis on coding, debugging, and testing of programs.

6.4 The Analyst

The analyst performs validation, verification, and testing (VV&T) which is a process of review, analysis, and testing that is employed throughout the software life cycle to ensure the production of

quality software [FIPS101]. When a team member is in the role of an analyst, development of software is not done. Instead, analysis of what has been developed is done. For example, requirements specifications are checked for consistency and completeness, software designs and programs are validated to insure that they meet the requirements, programs are verified to insure that they implement the design, test cases are checked for completeness, and test results are checked for correctness. All errors that are discovered are recorded and reported to the appropriate team member.

The tools and techniques of the analyst are described in NBS Special Publication 500-93 [NBS93], which is a technique and tool reference guide for software VV&T. The tools and techniques include:

algorithm analysis
analytic modeling
assertion generation
cause-effect graphing
code auditor
comparator
control structure analyzer
cross-reference generator
data flow analyzer
execution time analyzer
formal reviews
round-off analysis
inspections
interactive test aids
interface checkers
mutation analysis
peer review
units checking
regression testing
requirements analyzer
requirements tracing
software monitor
functional testing
symbolic execution
test coverage analysis
test data generators
test support facilities
walkthroughs.

Although many of the tools in this list are discussed in Section 5, some, such as walkthroughs, peer review, and inspections, are methodologies. It is important for an environment to provide basic support for methodologies as well as analysis tools.

6.5 The Documentation Editor

By necessity, the technical director, designers, programmers, and analysts create documentation. Often, these team members fill the role of the documentation editor whether or not they have good writing skills. This can have a significant affect on the acceptance of the software system that is developed by the team, particularly if user documentation has a high level of technical jargon in it and the users are not versed in the area. This leads to some of the same issues that were discussed in Section 4.1 regarding on-line assistance for users.

Therefore, if the team has a member with good writing skills that can take the role of documentation editor, the system developed is more likely to be acceptable to the users. Another advantage is that system documentation will also be much clearer, e.g., the programmer gets better design documentation, the maintainer gets a better maintenance manual. Since the documentation editor needs to understand the system being documented, it is also possible that the documentation editor will discover parts of the system that are weakly specified and need further work.

The tasks of the documentation editor include the following:

- rework documentation
- add references
- insure consistency
- oversee the mechanics of production.

To fulfill these tasks, the documentation editor needs office automation capabilities. In particular, editing, word processing, text formatting, and document preparation facilities are important to the documentation editor.

The UNIX environment contains a number of capabilities that are useful to the documentation editor. The "me" macro package provides ways to declare logical parts of a document, such as titles, authors, abstract, table of contents, section headings, paragraphs, and index. In addition to the "me" macro package, UNIX includes a tool called "eqn" for mathematical expressions and "tbl" for tables. All of these tools combine to make documents that are typesetter quality.

Other document preparation tools found in UNIX include:

refer	- find and insert literature references in documents
pic	- generate graphics (boxes and arrows) in documents
style	- analyze the writing style of a document
diction	- print wordy sentences
explain	- thesaurus
spell	- report possible spelling errors.

6.6 The Librarian

Section 3.1 states that the database is the repository where tools in the environment deposit output or get input. Because the development of a software system can generate a lot of information for the database, a librarian is needed to maintain the database. It is the librarian's job to purge the database of inactive information and to insure there are ample backup copies. If the inactive information is important, then the librarian archives it on tape and/or stores it away in printed hardcopy.

The librarian is also responsible for maintaining the product library. The product library is essentially a hard copy version of the most important parts of the database. It contains the most recent versions of each product generated by the team. It should include specifications, designs, programs, test data, and all associated documentation.

To maintain the database and the product library, the librarian must have archiving and cataloging facilities in the environment. In addition to these facilities, the librarian also needs recovery facilities should parts of the database be accidently deleted or otherwise lost. Finally, to recognize inactive parts of the database, the librarian needs database access histories and usage summaries.

6.7 The Maintainer

The software maintainer performs those activities required to keep a software system operational and responsive after it is accepted and placed into production [FIPS106]. There are three maintenance categories [NBS106]:

> Perfective maintenance - includes all changes, insertions, deletions, modifications, extensions, and enhancements which are made to a system to meet the evolving and/or expanding needs of the user.
>
> Adaptive maintenance - consists of any effort which is initiated as a result of changes in the environment in which a software system must operate.
>
> Corrective maintenance - refers to changes necessitated by actual induced or residual errors in a system.

The process of implementing a change to a software system requires a set of steps that is similar to the original development of software. In other words, maintenance has a life cycle similar to the one presented in Section 2.1. The number of steps in the maintenance process depends on the magnitude of the change. If the change is the correction of a residual error in the system, then it is unlikely that the design of the system will be changed. However, if the change is a major enhancement, then the maintenance process should include all of the following steps:

1. Determination of need for change
2. Submission of change request
3. Requirements analysis
4. Approval/rejection of change request
5. Scheduling of task
6. Design analysis
7. Design review
8. Code changes and debugging
9. Review of proposed code changes
10. Testing
11. Update documentation
12. Standards audit
13. User acceptance
14. Post installation review
15. Completion of task.

Judging from the previous list of steps, one can see that the maintainer should have access to the same software engineering environment as the original developers had. Unfortunately, in many cases this is not the case. Often software is turned over to a maintenance organization and the environment remains behind with the developers. The US Department of Defense has recognized this problem and one of the goals of the APSE is to eliminate it. If the APSE is GFE (government furnished equipment), the environment used by the developers and maintainers will be the same.

7.0 SUPPORT FOR APPLICATION

Some environments, e.g., programming environments, are oriented to specific types of users on a software development team. This section continues the discussion of environments that have a specific orientation; but the orientation will be to a specific type of target software. That is, the environment is specifically oriented toward developing software that is for a certain application.

7.1 Systems Development

A system is an integrated whole that is composed of diverse, interacting, and specialized structures and subfunctions [IEEE729]. The parts of a system, i.e., subfunctions, or the entire system can consist of hardware, software, or even people. Environments, which themselves may be referred to as systems, that deal with the development of systems provide capabilities that allow the characterization of the subsystems no matter what they consist of. In order to provide these capabilities, systems development environments deal with a system at a very high level and can be in most cases classified as framing environments. Two examples of systems development environments, DREAM and SARA, are discussed in the section on framing environments, Section 2.4.

Another example of a systems development environment that was briefly mentioned in Section 3.3.5 is the Automated Interactive Simulation System (AISIM) [Aust82]. AISIM is applicable to the design and analysis of proposed systems as well as to the operational analysis of existing systems. The characteristics of systems that can be modeled using AISIM include procedural operations, parallel processing, shared resources, operational loading, process communication, and interconnected networks. To build a model, AISIM provides a set of "entities" that are used to describe the system in terms which are understood by the simulation system. The following are examples of AISIM entities:

Process - a description of the logic of the operations, decisions, or activities of the system being modeled
Primitive - logical function that is used to define a process
Item - a transient data element such as a message
Queue - an ordered holding area for items
Resource - an object necessary to perform a process
Scenario - the environments in which a system must perform
Load - the affects on the system by the outside world.

Other entities include constants, variables, and tables.

AISIM is a command oriented system. At the highest level (the READY level), the user can issue commands which invoke certain system functions or descend to a lower level. If the user descends to a lower level, then a new set of commands becomes available. Examples of lower levels include the design level and the analysis level. At the design level, commands are available to define the system model by creating, modifying, or deleting entities. This level includes two editors: (1) a process editor which allows the user to describe the logical flow of a process graphically and (2) an architecture design editor which allows the user to define the layout and interconnection of the physical aspect of a network including its various processes.

At the analysis level, the user can exercise the model by simulation. This level includes commands which allow the user to modify scenarios and loads to see the effects they have on the system being modeled. In addition, commands are available to view the outputs of the simulation.

7.2 Embedded Systems

Embedded systems are systems that are an integral part of larger systems. The key characteristic of environments that support the development of embedded systems is that they allow the development of software for machines i.e., target machines, that may not be present in the development environment. Two examples of environments for developing embedded systems are FASP and APSE. FASP was briefly described in Section 3.3 and APSE was briefly described in Section 3.1. FASP is a general environment and APSE is a programming environment.

7.3 Information Systems

Environments that support the development of information systems provide capabilities for developing software that must process and manage large amounts of data. In addition, information systems must communicate the data that they contain. Therefore, addressing human factors is very important in information systems development.

Because human factors play such an important role in information system development, environments should contain tools that allow rapid development of prototypes so that feedback from the end user of the information system can be obtained early in development. In particular, environments should support [Blum82]:

Screen and report formatting - capabilities to produce a set of user interfaces which resemble those of the final system.

Partial and incomplete implementation - features that allow the identification and implementation of a subset of the total system.

Selective implementation - features such as screen managers, data base managers, and report generators that allow the development of specific components of the total system.

Each of the above features allow the user to quickly piece together parts of an information system.

An example of an environment that supports the development of information systems and that includes rapid prototyping tools is USE [Wass82] (See Section 4.3.4).

7.4 Data Processing Applications

Environments that are oriented toward the development of data processing (DP) applications are a popular topic in DP literature [Mart82] [Rin82] [Zoll80]. These environments are often called application generation or program generation systems [Blum82]:

Application generation system - an interpretive system that is molded to a specific DP environment. A user of the system types in a specification of the application desired and the system responds by interpreting the specification and performing the desired function.

Program generation system - provides the same capability as an application generation system but instead of interpreting the user's request, a program written in a language such as COBOL, PL/I, or MUMPS is produced that performs the desired function.

Typical functions that are generated include database management and update, report generation, retrievals, graphics, statistical analysis, and screen layouts.

A simple example of a specification that generates a table that summarizes sales information by account number might be:

```
SUM SALES
BY ACCOUNT
TABLE
```

Simple specifications can usually be provided to a generation system in any order, thus making the specification non-procedural. For more complex applications, such as screen control or the development of complex reports, the user would be required to use procedural constructs. In cases such as this, the application generation environment may prove to be more cumbersome then traditional software development environments.

7.5 Security-Critical Applications

Development of systems for security-critical applications such as defense communications requires special techniques. Environments have been developed within the research community to support some special, formal techniques. Formal techniques support mathematical proof of consistency between a specification of a system and a lower level specification of the same system, e.g., a model specification and a high level design specification or mathematical assertions about a program and the program itself.

Formal verification increases confidence that the lower level specification is correct because it symbolically evaluates and "tests" the lower level specification with many more cases than normal testing can. However, even with the special tools that have been developed, formal verification is still so labor intensive that it has previously been used only for special defense applications, such as the development of "secure" operating systems [Perr84].

Three examples of environments that provide verification features are summarized by London and Robinson [Lond80]:

Affirm

The Affirm system developed at the USC Information Sciences Institute, is for the algebraic specification and verification of abstract data types and Pascal-like programs which use these types in expressions and assertions. A natural deduction theorem prover uses powerful rewrite rule facilities and user-directed proof steps to prove program verification conditions and properties of data types. Additional features include tools for the analysis of algebraic specifications and a library of data types. Experiences include the specification and partial proof of a large file updating module and the proof of several high-level properties in the application areas of protocols and security kernels.

Gypsy

The Gypsy system is located at the University of Texas at Austin. Gypsy is a language for both specifying and implementing programs. Important applications to date have been for communication systems, for which Gypsy has special-purpose language constructs and proof rules. The verification system maintains the complete state of a system as it is being developed (both specifications and implementation). If any part of a system is changed, the system can identify the proofs that must be redone. This incremental approach reduces much of the effort in verifying programs.

HDM

The verification system at SRI International is based on the Boyer-Moore theorem prover and on HDM (Hierarchical Development Methodology). The Boyer-Moore theorem prover proves theorems in a theory based on recursive functions and inductively defined objects. HDM is a methodology for formally specifying, implementing, and verifying programs. The computational model of HDM is a hierarchy of abstract machines. The verification system is intended to be used with many different implementation languages. It currently works for Modula and a subset of Jovial J73.

References for the three systems include: [Thom81], [Ambl77], and [Robi77].

8.0 HARDWARE SUPPORT

Stucki formulates the software paradox [Stuc83]:

The software community has done an excel-

lent job of attempting to automate everyone's job except their own.

The software engineering environments that have been discussed in this report provide some evidence that the "software community" is attempting to automate. However, in terms of the use of hardware, the paradox still lives on. Most software engineering environments use traditional hardware, i.e., common line-oriented terminals and printers attached to mainframe computers. Meanwhile, innovators are taking advantage of the hardware "revolution" for other applications and providing systems that use graphics, voice, pointing devices, workstations, and microcomputers. Applications for which this is particularly true include computer aided VLSI design tools, systems for the handicapped, automobile design, office automation, airplane design, and personal computer applications.

8.1 Hardware Selection Issues

There are several reasons why software engineering environments use traditional hardware. One reason is portability. Building or buying a software engineering environment is a major investment and tying it to any particular hardware may limit its potential usefulness which thereby may limit the received benefits to an organization. Consequently, developers of software engineering environments tend to choose hardware conservatively.

Another reason for using traditional hardware is methodology. Because software engineering is still an emerging field, the disciplines are not yet well defined. In order to take advantage of hardware features such as graphics, the environment developer must know what graphics primitives to provide in the environment. Since there is little agreement between methodologies on what primitives are necessary or what they should look like, the environment developer must either generalize or choose a specific methodology which once again may limit an environment's potential benefit.

The need for minimal communication between the host computer and the users has also impacted the use of traditional hardware. Because host computers have to service many users, it is important that communications between the host and any single user be kept as low as possible. This is usually referred to as a low bandwidth between the host and the users. To take advantage of modern hardware, such as bit-mapped terminals that provide color graphics and multiple windows, pointing devices, and voice input and output devices, a higher bandwidth is needed between the user and the host computer. Normally, to obtain this higher bandwidth, a computer processor is dedicated to a single user. An example of this is presented in the next section.

8.2 Benefits Gained by Selecting and Dedicating Hardware

If the environment developer chooses specific hardware and dedicates a computer processor to a single user, the results can be dramatic. An example of this is the Symbolics 3600 Lisp Machine. The Symbolics 3600 Lisp Machine is a single user environment that not only uses specific hardware, but also standardizes on the Lisp programming language and the methods associated with its use. Some of the features provided by the Symbolics 3600 include the following:

- mouse pointing device with three function buttons
- multi-window screen displays (including a mouse documentation line and a status line)
- color graphics
- window scrolling
- menus
- file handling
- networking capabilities
- a message facility and a mail system
- help keys (for obtaining on-line assistance)
- a Lisp subenvironment that includes:
  - Zmacs, to create Lisp source code and to compile functions and files (a language oriented screen editor that includes its own help system and windowing operations)
  - Lisp Listener, to run Lisp code
  - Debugger, to examine the Lisp Environment (a highly interactive debugger)
  - Inspector, to inspect and modify Lisp data structures.

Because the Symbolics 3600 is single-user, it provides much higher performance when compared to Lisp Environments on shared mainframe computers.

As the above example shows, choosing specific hardware allows the developer to take full advantage of the hardware's capabilities. The environment developer does not have to design the environment around traditional hardware and a high bandwidth can be provided between the computer processor and the user.

8.3 Workstations

One way to breach the hardware gap is to incorporate workstations with dedicated processors into the environment. If the user has a workstation instead of a standard terminal, then theoretically a high band-width can be provided between the workstation and the user while maintaining a low bandwidth between the workstation and the host computer.

Gutz, Wasserman, and Spier [Gutz81] have proposed a similar type of environment. The workstations in this proposed environment have the following features:

- 1 megabyte memory
- 32 bit computer processor
- graphics capability
  - multiple-character fonts
  - reverse video
  - variable intensity
  - multiple windowing capability
  - color
  - full-page text display
- 40 megabyte hard disk
- standard floppy disk
- audio input/output
- pointing device (mouse, tablet, or light pen).

The workstations communicate with one or more host computers which provide archiving, database, and other input/output services. However, all this capability may not be necessary. Many organizations are using personal computers, such as the IBM PC, to off-load the demand on their mainframes and minicomputers. It may be possible in certain circumstances to incorporate personal computers as workstations in an environment.

Management of documentation and code that is distributed among several workstations is not a trivial task. Leblang and Chase [Lebl84] propose the following set of five "managers", part of what they call a distributed workstation environment, to handle this problem:

(1) a history manager to maintain source code control

(2) a configuration manager to maintain program and sub-program relationships and to configure the system being developed

(3) a task manager to relate source changes to higher level activities

(4) a monitor manager to watch for user-defined dependencies

(5) an advice manager to disseminate general project information.

9.0 LEVELS OF SUPPORT

Previous sections discussed the types of environments, the features they provide, and the factors that affect their development. Another factor that impacts their development is the intended level of support that can be provided by an environment. This section presents three classifications for levels of support where each has a slightly different orientation and concludes with a list of generic capabilities that are important for all environments to have.

9.1 Levels of Support Based on Project Size

A paper by Howden [Howd82] proposes a four level classification that is life cycle oriented and includes increased use of tools and techniques based on project size. Howden assumes that standard operating system features, such as compiling, editing, debugging, and file management, are part of each environment. Although it is not stated in the paper, the items which are considered "unsupported" can be supported by these standard features. Howden's four level classification is presented in Table 9.1.

A recent survey by Hecht [NBS82] shows that as project size increases so does the use of tools. Therefore, Howden's four levels based on project size parallels current practice.

Medium projects were assumed to have the following characteristics: 2 year development time, $2,000,000 project budget, 15-20 year system lifetime, sophisticated users, and a staff of 7 programmers and 1 manager plus additional support staff. While large projects were assumed to have these characteristics: 3-5 year development time, $20,000,000 project budget, 10 year system lifetime, unsophisticated users, and a staff of 70 programmers and 5-7 managers plus additional support staff.

The Level I environment was considered to contain the minimum set of tools and methods without which it would be "foolish" to attempt to carry out a medium scale development. The Level II environment contains tools and methods for assisting the users in the more important parts of software development. Level III requires that the Level II tools be compatible. Level IV is an elaboration of the Level III tools with features to handle large scale development including an integrated archiving facility.

9.2 Levels of Support Based on Capability

Branstad, Adrion, and Cherniavsky [Bran81] propose levels of support that are based strictly on increased levels of tool capability. The levels, which are presented in Table 9.2, provide a way of gauging the level of support provided by an environment.

The Minimal System (D1) contains features common to most operating systems. The Basic System (D2) augments D1 with a database and features to support the management of code and documentation. "Make", which is included in D2, is a capability found in UNIX which configures programs or documentation [Feld79]. The Full System (D3) completes coverage for the entire life-cycle. The Advanced System (D4) is claimed to be more a goal then a reality. It fully integrates all the features of D2.

	Level I	Level II	Level III	Level IV
project size	medium	medium	medium and large	large
cost	$35,000	$200,000	$300,000	$3,000,000
requirements tools and techniques	unsupported data flow diagrams or structure charts	supported data flow diagrams or structure charts	supported data flow diagrams or structure charts	supported data flow diagrams or structure charts
design tools and techniques	unsupported design, automated data dictionary	supported design, automated data dictionary	supported design, automated data dictionary	supported design, automated data dictionary
coding tools	simple source code control	source code control, configuration management tools	source code control, configuration management tools	source code control, configuration management tools
verification tools and techniques	comparator, unsupported test plan and test data generation	comparator, coverage analyzer, test harness	comparator, coverage analyzer, test harness	comparator, coverage analyzer, test harness
management tools and techniques	manual milestone or Gantt charts	automated project control	automated project control	automated project control
other features			compatible tools, database	compatible tools, database, integrated archiving facility

Table 9.1 Levels of Support Based on Project Size

	Standard Level Features	Additional Support for Critical Applications
D1 - Minimal System	Translation Cross-Reference Trace Audit Optimization File Comparison Text Editing	Range Checking Type Analysis Assertion Checking Formatting
D2 - Basic System	D1 Information Repository Separate Compilation Make Interface Analysis Version Control Data Dictionary Test Coverage	Data Flow Analysis Structure Analysis Complexity Measurement Performance Monitor
D3 - Full System	D2 Requirements Specification Requirements Analysis Design Specification Design Analysis Test Harness Automated Documentation Automated Project Control	Symbolic Evaluation Proof of Correctness
D4 - Advanced System	D3 with Information Interfaces Specified and Full Integration	

Table 9.2 Levels of Support Based on Capability

	Management, Transformation, and Input/Output Features	Static Analysis and Dynamic Analysis Features
Required	Configuration Control Ada Library Management Formatting Compilation Optimization Editing Command Assistance Error Assistance	Type Analysis Cross Reference Tracing/Debugging Interface Analysis
Important	Specification Management Data Dictionary Management Test Management Cost Estimation Scheduling Tracking Syntax-Directed Editing Menu Assistance Auditing	Structure Checking Reference Analysis Timing Analysis Tuning Analysis Regression Testing Coverage Analysis
Useful	Ada Package Management On-Line Tutor Definition Assistance	Statistical Profiling Complexity Measurement Completeness Checking Consistency Checking Assertion Checking

Table 9.3 Levels of Support by User Priorities

9.3 Levels of Support Based on User Priorities

During the development of a report for the Ada Joint Program Office that defines a taxonomy of tool features for the Ada environment [NBS2625], reviewers were asked to prioritize the features in the taxonomy. The priorities were to be established from a user's perspective at four levels: required, important, useful, and unnecessary. None of the tool features was placed in the last category. Table 9.3 lists the features in each of the remaining three categories.

9.4 Generic Support

Most current environments would not measure well in terms of complete coverage as defined in Sections 9.1, 9.2, and 9.3. This is especially true for programming and framing environments because they do not emphasize complete life cycle coverage. Even Ada environments that either are available today or are being developed do not completely match the base-level "required" features presented in Section 9.3. Part of the problem is that environments are still found primarily in research organizations and have yet to make the transition to common practice.

Many research questions must be answered before environments can make that transition to common practice. However, even without a consensus on the support that should be provided by an environment, it is possible to define the generic capabilities that are important for all environments to have. These are:

editing - to support the development of documentation, programs, and project communications.

compilation - to support the translation of program to lower level languages.

formatting - to support the development of documentation and more formal project communications.

debugging - to support the dynamic analysis of programs.

integrated user interface - to promote user productivity and user acceptance of the environment.

database - to support information management, version control, and maintenance.

These capabilities are referenced consistently throughout this report, e.g., in examples of existing environments, among development issues, and finally, when in the levels of support.

10.0 CONCLUSION

This report has presented information, with examples, that characterizes and describes software engineering environments. Software engineering environments provide software tools to aid in the systematic development and maintenance of all the products associated with software systems; hence the report has made frequent reference to software engineering environments as systems. Existing and emerging software engineering environments provide such tools with varying degrees of interdependence among them, for differing tasks and users, with different levels of completeness or concentration.

In this report the software engineering environments have been discussed within the framework of a complete software development and maintenance life cycle, in which iteration or repetition through an entire or partial cycle, may occur. The framework also clarifies the relationships among the many tasks of software engineering and thus places emphasis on broad categories of tasks that can be supported by automation. Some of these categories include the following:

- management of software projects
- development, maintenance of software systems
- improvement of the quality of software products
- increased productivity of project personnel.

The examples in this report indicate that many environments provide some software project management assistance but none yet provide all the tools necessary to plan, schedule, monitor and control over the entire software life cycle. Similarly, within the technical tasks, most environments tend to provide generic support for most tasks or to focus heavily on certain types of tasks, such as those for designing a software system or for programming and analyzing a system. Hence, examples in this report refer to "framing environments" or "programming environments." Again, no examples were found that provide comprehensive technical coverage of all tasks expected to occur in development and maintenance although several are comprehensive for their areas of concentration. Also, almost every type of task that is performed within software engineering can be found in at least one environment.

Many of the environments provide service to analyze the software, either statically or dynamically. Some of these tools can be applied to early software products, such as requirements or design specifications. Other tools help to organize test libraries or software configurations. All of these contribute to the quality of the software products.

This report has addressed the requirements to make a software engineering environment readily usable by software engineers of varying levels of expertise. Examples portrayed different kinds of on-line help systems and user features that contribute to user acceptance of a software engineering environment. Only after acceptance and usage can a software engineer reap the potential benefits of software engineering environments.

This report has also shown that various other factors may affect the design and capabilities of the software engineering environment. Some of these include hardware considerations and the type of applications the environment will be used to develop. The target computer's processing units might impact operating speed of some of the functions the environment is expected to perform. Also, hardware accessories, such as a pointing device for command control, may drive the design of some environments. Speedy transmission of vast amounts of data for an information system may require a different type of environment for its development from an environment intended to be used in developing an embedded system.

This report has defined generic capabilities that are important for all environments to have. There are varying levels of support that add on to these capabilities. The types of support are frequently tailored to satisfy a specific group of users. Even if a complete environment with the highest support level were developed, it is likely that it would serve only a small part of the software engineering community.

11.0 REFERENCES

[ACM82]
"Special Issue on Rapid Prototyping," Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop, *Software Engineering Notes* , Vol. 7, No. 5, December 1982.

[ACM84]
"Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments," *Software Engineering Notes* , Vol. 9, No. 3, May 1984.

[Alfo81]
M. W. Alford and C. G. Davis, "Experience with the Software Development System," *Software Engineering Environments* , H. Hunke, Editor, North-Holland, 1981.

[Ambl77]
A. L. Ambler, et al, "Gypsy: A Language for Specification and Implementation of Verifiable Programs," *Software Engineering Notes* , March 1977.

[Aust82]
W. P. Austell, Jr., M. D. Deshler, and J. W. Hearne, "Automated Interactive Simulation Model (AISIM)," CDRL 104, Contract No. F19628-79-C-0153, Hughes Ground Systems Group, Fullerton, CA, February 1982.

[Bass83]
P. Bassett and J. Giblon, "Computer Aided Programming (Part I)," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[Baye81]
M. Bayer, et al, "Software Development in the CDL2 Laboratory," *Software Engineering Environments* , H. Hunke, Editor, North-Holland, 1981.

[Blum82]
B. Blum and R. Houghton, "Rapid Prototyping of Information Management Systems," Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop, *Software Engineering Notes*, Vol. 7, No. 5, December 1982.

[Booc83]
Grady Booch, "Object-Oriented Design," *Tutorial on Software Design Techniques*, Fourth Edition, IEEE No. EHO205-5, 1983.

[Boeh78]
B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod and M. J. Merritt, *Characteristics of Software Quality*, North-Holland Publishing Company, NY, 1978.

[Boeh81]
Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[Boeh84]
Boehm, Barry W. et al, "A Software Development Environment for Improving Productivity," *Computer* , Vol. 17, No. 6, June 1984.

[Bran81]
Martha A. Branstad, W. Richards Adrion, and John C. Cherniavsky, "A View of Software Development Support Systems," *Proceedings of National Electronics Conference*, Chicago, IL, October, 1981.

[Broo75]
F. P. Brooks, Jr, *The Mythical Man-Month*, Addison-Wesley Pub. Co., 1975.

[Buxt80]
J. Buxton, "Requirements for Ada Programming Support Environments: STONEMAN," U.S. Department of Defense, Washington, DC, February 1980.

[Bryc83]
M. Bryce, " PRIDE - Automated Systems Design Methodology," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[Cain75]
S. H. Caine and E. K. Gordon, "PDL: A Tool for Software Design," *Proceedings of the National Computer Conference*, 1975.

[CAIS83]
"Draft Specification of the Common APSE Interface Set (CAIS), Version 1.1," KIT/KITIA CAIS Working Group for the Ada Joint Program Office, NTIS AD-A134825, September 1983.

[Clar81]
I. A. Clark, "Software Simulation as a Tool for Usable Product Design," *IBM Systems Journal*, Vol. 20, No. 2, 1981.

[Cowe83]
Wayne R. Cowell, and Leon J. Osterweil, "The Toolpack/IST Programming Environment," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[DACS79]
"Quantitative Software Models," Data and Analysis Center for Software, SRR-1, March 1979.

[Deli84]
Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz, "Viewing a Programming Environment as a Single Tool," *Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on*

Practical Software Development Environments, Gaithersburg, MD, April 1984.

[Estr78]
G. Estrin, " A Methodology for Design of Digital Systems - Supported by SARA, Age: 1," *Proceedings of the National Computer Conference*, June 1978.

[Fair78]
R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software," *Computer*, April 1978.

[Feld79]
S. I. Feldman, "Make - A Program for Maintaining Computer Programs," *Software - Practice and Experience*, Vol. 9, 1979.

[Fenc81]
R. Fenchel, "An Integral Approach to User Assistance," *Proceedings of the Conference on Easier and More Productive Use of Computer Systems*, ACM Order No. 608811, May 1981.

[Fenc82]
R. S. Fenchel and G. Estrin, "Self-Describing Systems Using Integral Help," *IEEE Transactions on Systems, Man, and Cybernetics*, March-April 1982.

[FIPS38]
Guidelines for Documentation of Computer Programs and Automated Data Systems, National Bureau of Standards FIPS PUB 38, February 1976.

[FIPS64]
Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase, National Bureau of Standards FIPS PUB 64, August 1979.

[FIPS76]
Guideline for Planning and Using a Data Dictionary System. National Bureau of Standards FIPS PUB 76, August 1980.

[FIPS99]
Guideline: A Framework for the Evaluation and Comparison of Software Development Tools, National Bureau of Standards FIPS PUB 99, March 1983.

[FIPS101]
Guideline for Lifecycle Validation, Verification, and Testing of Computer Software, National Bureau of Standards FIPS PUB 101, June 1983.

[FIPS106]
Guideline on Software Maintenance, National Bureau of Standards FIPS PUB 106, June 1984.

[Fisc84]
C. N. Fischer, et al, "The Poe Language-Based Editor Project," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Gaithersburg, MD, April 1984.

[Free83a]
Peter Freeman and Anthony I. Wasserman, "Ada Methodologies: Concepts and Requirements," *ACM Sigsoft Software Engineering Notes*, Vol. 8, No. 1, January 1983.

[Free83b]
Peter Freeman, "The Fundamentals of Design," *Tutorial on Software Design Techniques*, Fourth Edition, IEEE No. EHO205-5, 1983.

[FSWE80]
Federal Software Exchange Catalog, General Services Administration, GSA/ADTS/ C-80/3, FSWEC-80/0118, September 1980.

[GAO80]
"Wider Use of Better Computer Software Technology Can Improve Management Control and Reduce Costs," Comptroller General's Report to the Congress, U.S. General Accounting Office, FGMSD-80-38, April 29, 1980.

[Gunn59]
R. Gunning, *How to Take the Fog Out of Writing*, Dartnell Press, Inc., Chicago, 1959.

[Gutz81]
Steve Gutz, Anthony I. Wasserman, and Michael J. Spier, "Personal Development Systems for the Professional Programmer," *Computer*, Vol. 14, No. 4, April 1981.

[Hall80]
Dennis E. Hall, Deborah K. Scherrer, and Joseph S. Sventek, "A Virtual Operating System," *Communications of the ACM* , Vol. 23, No. 9, September 1980.

[Hals77]
M. H. Halstead, *Elements of Software Science*, Elsevier - North Holland Pub. Co., New York, 1977.

[Howd78]
W. E. Howden, "A Survey of Static Analysis Methods," *Tutorial: Software Testing and Validation Techniques*, IEEE Cat. No. EHO138-8, 1978.

[Howd78a]
W. E. Howden. "A Survey of Dynamic Analysis Methods," *Tutorial: Software Testing and Validation Techniques*, IEEE Cat. No. EHO138-8, 1978.

[Howd82]
William E. Howden, "Contemporary Software Development Environments," *Communications of the ACM* , Vol. 25, No. 5, May 1982.

[Houg84]
R. Houghton, "Online Help Systems: A Conspectus," *Communications of the ACM* , Vol. 27, No. 2, February 1984.

[ICSE81]
Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81-CH1627-9), March 1981.

[ICSE82]
Proceedings of the 6th International Conference on Software Engineering, (IEEE Order No. 82-CH1795-4), September 1982.

[ICSE84]
Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84-CH2011-5), March 1984.

[IEEE729]
IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 729-1983.

[Inte82]
"System Specification for Ada Integrated Environment, Type A," Intermetrics, Inc., IR-676-2, November 1982.

[Jack75]
M. A. Jackson, *Principle of Program Design,* Academic Press, 1975.

[Kern76]
B. Kernighan and P. Plauger, *Software Tools,* Addison-Wesley Pub. Co., 1976.

[Kern81]
Brian W. Kernighan and John R. Mashey, "The UNIX Programming Environment," *Computer* , Vol. 14, No. 4, April 1981.

[Knut71]
D. Knuth, "An Empirical Study of FORTRAN Programs," *Software-Practice and Experience* , 1971.

[Lebl84]
David B. Leblang and Robert P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Gaithersburg, MD, April 1984.

[Lecl82]
Y. Leclerc, S. W. Zucker, and D. Leclerc, "A Browsing Approach to Documentation," *Computer* , June 1982.

[Lond80]
R. L. London and L. Robinson, "The Role of Verification Tools and Techniques," *Software Development Tools* by W. E. Riddle and R. E. Fairley, Springer-Verlag, 1980.

[Love83]
Tom Love, "Experiences with Smalltalk-80 for Application Development," *Proceedings of SoftFair,* (IEEE Order No. 83CH1919-0), July 1983.

[Lyon81]
M. J. Lyons, "Salvaging Your Software Asset," *Proceedings of the National Computer Conference,* May 1981.

[Mart82]
James Martin, *Application Development Without Programmers,* Prentice-Hall, 1982.

[McCa76]
T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering,* Vol SE-2, December 1976.

[Metz83]
J. J. Metzger and A. Dniestrowski, "Platine: A Software Engineering Environment," *Proceedings of SoftFair,* (IEEE Order No. 83CH1919-0), July 1983.

[Mill56]
G. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information," *Psychological Review* , Vol. 63, 1956.

[NBS3]
B. Leong-Hong and B. Marron, "Technical Profile of Seven Data Element Dictionary/ Directory Systems," National Bureau of Standards, NBS SP 500-3, February 1977.

[NBS78]
Martha A. Branstad and W. Richards Adrion, Editors, "NBS Programming Environment Workshop Report," National Bureau of Standards, NBS SP 500-78, June 1981.

[NBS80]
R. Houghton, Editor, *Proceedings of the NBS/IEEE/ACM Software Tool Fair,* National Bureau of Standards, NBS SP 500-80, October 1981.

[NBS82]
Herbert Hecht, "Final Report: A Survey of Software Tools Usage," National Bureau of Standards, NBS SP 500-82, November 1981.

[NBS88]
R. Houghton, "Software Development Tools," National Bureau of Standards Special Publication 500-88, March 1982.

[NBS93]
Patricia B. Powell, Editor, "Software Validation, Verification, and Testing Technique and Tool Reference Guide," National Bureau of Standards, NBS SP 500-93, September 1982.

[NBS106]
Roger J. Martin and Wilma M. Osborne, "Guidance on Software Maintenance," National Bureau of Standards, NBS SP 500-106, December 1983.

[NBS359]
"Fortran 77 Analyzer User Manual," National Bureau of Standards, NBS GCR 81-359, 1981.

[NBS418]
M. Shadad and E. Libster, "Compiler Features: A Survey," National Bureau of Standards, NBS GCR 82-418, December 1982.

[NBS2625]
Raymond C. Houghton, Jr., "A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE)," National Bureau of Standards, NBSIR 82-2625, February 1983.

[NBS3113]
Raymond C. Houghton, Jr., "Annotated Bibliography of Recent Papers on Software Engineering Environment", National Bureau of Standards, NBSIR 85-3113, February 1985.

[Oste76]
L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs," *Software-Practice and Experience* , October 1976.

[Parn72]
D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* , December 1972.

[Perr84]
T. Perrine, J. Codd, and B. Hardy, "An Overview of the Kernelized Secure Operating System (KSOS)," 7th DoD/NBS Computer Security Conference, National Bureau of Standards, September 1984.

[Porc83]
Maria Porcella, Peter Freeman, and Anthony I. Wasserman, "Ada Methodology Questionnaire Summary," *ACM Sigsoft Software Engineering Notes*, Vol. 8, No. 1, January 1983.

[Pric81]
L. Price, "Using Offline Documentation Online," *Proceedings of the Conference on Easier and More Productive Use of Computer Systems*, ACM Order No. 608811, May 1981.

[Priv82]
J. P. Privitera, "Ada Design Language for the Structured Design Methodology," *Proceedings of the AdaTEC Conference*, October 1982.

[Raxo80]
R. Raxouk and G. Estrin, "Modeling and Verification of Communication Protocols in SARA: X.21 Interface," *IEEE Transactions on Computers* , December 1980.

[Rell81]
N. Relles, N. Sondheimer, and G. Ingargiola, "Recent Advances in User Assistance," *Proceedings of the Conference on Easier and More Productive Use of Computer Systems*, ACM Order No. 608811, May 1981.

[Rell81a]
N. Relles and L. A. Price, "A User Interface for Online Assistance," *Proceedings of the 5th International Conference on Software Engineering*, March 1981.

[Rell81b]
N. Relles and L. A. Price, "Sample Output: A User Interface for Online Assistance," *Proceedings of the NBS/IEEE/ACM Software Tool Fair*, NBS Special Publication 500-80, R. Houghton, ed., October 1981.

[Ridd81]
W. E. Riddle, "An Assessment of Dream," *Software Engineering Environments* , H. Hunke, Editor, North-Holland, 1981.

[Ridd83]
William E. Riddle, "The Evolutionary Approach to Building the Joseph Software Development Environment," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[Rin82]
N. Adam Rin, "An Interactive Applications Development System and Support Environment," *Automated Tools for Information Systems Design*, H. Schneider and A. Wasserman, Editors, North-Holland, 1982.

[Robi77]
L. Robinson and K. N. Levitt, "Proof Techniques for Hierarchically Structured Programs," *Communications of the ACM* , April 1977.

[Roem82]
J. M. Roemer and A. Chapanis, "Learning Performance and Attitudes as a Function of the Reading Grade Level of a Computer-Presented Tutorial," *Proceedings of the*

Conference on Human Factors in Computer Systems, Washington DC Chapter of the ACM, March 1982.

[Roth79]
J. Rothenberg, "On-Line Tutorials and Documentation for the SIGMA Message Service," *Proceedings of the National Computer Conference*, 1979.

[Rube83]
Burt L. Rubenstein and Richard A. Carpenter. "The Index Development Environment Workbench," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[Saib81]
S. H. Saib, J. P. Benson, C. Gannon, and W. R. DeHaan. "RXVP80: A Software Documentation, Analysis, and Test System," *Proceedings of the NBS/IEEE/ACM Software Tool Fair*, NBS Special Publication 500-80, R. Houghton, ed., October 1981.

[Shne80]
Shneiderman, B., *Software Psychology: Human Factors in Computer and Information Systems* , Winthrop, Cambridge, Mass., 1980.

[Soft83]
Proceedings of SoftFair, A Conference on Software Development Tools, Techniques, and Alternatives, (IEEE Order No. 83CH1919-0), July 1983.

[Stay76]
J. F. Stay. "HIPO and Integrated Program Design," *IBM Systems Journal* , Vol. 15, No. 2, 1976.

[Steu84]
H. G. Steubing, "A Software Engineering Environment (SEE) for Weapon System Software," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1984.

[Stuc83]
Stucki, Leon G.,"What about CAD/CAM for Software? The ARGUS Concept," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[Tayl84]
Richard N. Taylor and Thomas A. Standish, "Steps to an Advanced Ada Programming Environment," *Proceedings of the 7th International Conference on Software Engineering*, (IEEE Order No. 84CH2011-5), March 1984. "[Teic77]" D. Teichroew and E. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation of Information Processing Systems," *IEEE Transactions on Software Engineering* , Vol SE-3, No 1, 1977.

[Teit81]
Warren Teitelman and Larry Masinter, "The Interlisp Programming Environment," *Computer* , Vol. 14, No. 4, April 1981.

[Teit81a]
T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, Vol. 24, No. 9, September 1981.

[Teit84]
W. Teitelman, "A Tour Through Cedar," *Proceedings of the 7th International Conference on Software Engineering*, (IEEE Order No. 84CH2011-5), March 1984.

[Thom81]
D. H. Thompson, et al, "Specification and Verification of Communication Protocols in Affirm," ISI/RR-81-88, USC/Information Sciences Institute, February 1981.

[UNIX42]
UNIX PROGRAMMER"S MANUAL, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94270, August, 1983.

[Warn74]
J. Warnier, *Logical Construction of Programs* , Van Nostrand Reinhold, 1974.

[Wass82]
A. Wasserman and D. Shewmake, "Rapid Prototyping of Interactive Information Systems," Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop, *Software Engineering Notes*, Vol. 7, No. 5, December 1982.

[Wass83]
Wasserman, Anthony I, "The Unified Support Environment: Tool Support for the User Software Engineering Methodology," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[Wolf81]
Martin I. Wolfe, et al, "The Ada Language System," *Computer* , Vol. 14, No. 6, June 1981.

[Your75]
E. Yourdon and L. Constantine, *Structured Design* , Yourdon Press, New York, 1975.

[Zajo83]
P. C. Zajonc and K. J. McGowan. "Proto-Cycling: A New Method for Application Development Using Fourth Generation Languages," *Proceedings of SoftFair*, (IEEE Order No. 83CH1919-0), July 1983.

[Zoll80]
M. L. Zollicker, Editor, "Proceedings of a Conference on Application Development Systems," ACM Order No. 473800, March 1980.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See Instructions)	1. PUBLICATION OR REPORT NO. NBSIR-85/3250	2. Performing Organ. Report No. 644	3. Publication Date September 19, 1985

4. TITLE AND SUBTITLE

Characteristics and Functions of Software Engineering Environments

5. AUTHOR(S)

Raymond C. Houghton, Jr, and Dolores R. Wallace

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered
research
10/1/84 - 9/19/85

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

National Bureau of Standards
Institute for Computer Sciences and Technology
Gaithersburg, MD 20899

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

As part of the program to provide information to Federal agencies on software tools for improving quality and productivity in software development and maintenance, data was collected on software engineering environments. Software engineering environments surround their users with software tools necessary for systematic development and maintenance of software. The purpose of this report is to characterize software engineering environments by type and by their relationship to the software life cycle and by their capabilities, limitations, primary users, and levels of support. This report provides examples of existing software engineering environments that are available commercially or in research laboratories with the features and characteristics they provide.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

framing environments; human factors; life cycle coverage; programming environments; software analysis; software engineering environments; software support, software tools.

13. AVAILABILITY

☒ Unlimited
☐ For Official Distribution. Do Not Release to NTIS
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES
44

15. Price